SOFTWARE RELIABILITY PREDICTION

DONALD A. STEEL

A thesis submitted in partial fulfilment of the requirements of the Council for National Academic Awards for the degree of Doctor of Philosophy.

MAY 1990

Dundee Institute of Technology in collaboration with
NCR (Manufacturing) Ltd., Self Service Systems Division.

NOTE on CONFIDENTIALITY

This Thesis should remain confidential for two years after publication (September 1992).

SOFTWARE RELIABILITY PREDICTION - DONALD A. STEEL

ABSTRACT

The aim of the work described in this thesis was to improve NCR's decision making process for progressing software products through the development cycle. The first chapter briefly describes the software development process at NCR, detailing documentation review and software testing techniques. The objectives and reasons for investigating software reliability models as a tool in the decision making process are outlined. There follows a short review of software reliability models, with the Littlewood and Verrall Bayesian model considered in detail. The difficulties in using this model to obtain estimates for model parameters and time to next failure are described. These estimation difficulties exist using the model on good data sets, in this case simulated failure data, and the difficulties are compounded when used with real failure data. The problems of collecting and recording failure data are outlined, highlighting the inadequacies of these collected data, and real failure data are analysed. Software reliability models are used in an attempt to quantify the reliability of real software products. The thesis concludes by summarising the problems encountered when using reliability models to measure software products and suggests future research into metrics that are required in this area of software engineering.

TABLE OF CONTENTS

		PAGE No.
CHAPTER 1	Introduction	1
	1.1 Standard Development Process	3
	1.1.1 Business Plan Phase	3
	1.1.2 Planning and Definition Phase	5
	1.1.3 Design Phase	6
	1.1.4 Implementation Phase	7
	1.1.5 Integration Phase	9
	1.1.6 Certification Phase	10
	1.1.7 Field Follow Phase	11
	1.1.8 Discontinuation Phase	11
	1.2 Test/Review Procedures	12
	1.2.1 Inspection Process	12
	1.2.2 Software Testing	14
CHAPTER 2	Objectives	18
CHAPTER 3	Background to Software Reliability Models ...	23
	3.1 The Littlewood - Verrall (L-V) Model	26
	3.2 Estimation problems for (L-V) Model	30
	3.2.1 Sampling Distributions	36
	3.3 Debugging Models	41
CHAPTER 4	Data Gathering	47
CHAPTER 5	Data Analysis and Estimation	52
	5.1 Basic Data Analysis and Graphical Techniques	52
	5.1.1 Failure Data	53
	5.1.2 Cumulative Failure Profile	63
	5.1.3 Time Between Failures	72
	5.2 Reliability Growth Tests	82
	5.3 The Littlewood - Verrall (L-V) Model	84

CHAPTER 6 Conclusions and Suggestions for Further Work 92

6.1 Reliability Models 92

6.2 Improving Data & Analysis 94

6.3 Improving the Process 97

6.4 Suggestions for Further Work 99

APPENDIX A Littlewood - Verrall (L-V) Model A-1

APPENDIX B Raftery's Reliability Growth Test B-1

APPENDIX C Validation of the Random Number Generator . C-1

APPENDIX D SMARTWARE Project routines D-1

REFERENCES R-1

CHAPTER 1 INTRODUCTION

Software products have grown in size and complexity over the last 25 years, and with them the role of the programmer has evolved from someone who produces code from ill defined requirements without documentation and with little thought for future maintenance and enhancements, into that of a software engineer concerned with process management and control.

The software engineer develops structured formal designs from well documented and complete requirement specifications, using automated software design tools and validates output at each stage of a well documented and consistent software development process, using static and dynamic test tools.

The software engineer does not yet exist in every software development company but with the increasing use of structured design methodologies, development tools and processes, and for companies survival, that day may not be far away. Current practice however still tends towards correction rather than prevention and the work described in this thesis investigates methods for predicting software product quality once failures have been identified and corrected.

Large, complex software products require to be developed using a standard development process if there is to be any

chance of a high quality product being produced. In the remainder of this chapter the standard development process adopted by NCR is discussed.

NCR Corporation is a multinational manufacturer of computer products, ranging from personal computers to large mainframes, computer peripherals and software. The markets are in the commercial, financial and retail sectors. The headquarters are in the U.S.A. but manufacturing, research and development plants are located worldwide.

The NCR plant at Dundee has development and manufacturing responsibility for all financial and retail terminals marketed by NCR Corp. (Self Service Systems); the major product range being financial automated teller machines (ATM). With these ATMs handling large amounts of currency there is a necessity for a high quality, highly reliable, secure product.

NCR Corp. are continually striving for high quality software products and to this end have set up an advanced quality system group within corporate quality assurance, to ensure that NCR remain in the forefront of software engineering science. This group is responsible for reviewing the latest literature, evaluating new commercial software tools, planning and organising software education courses and arranging internal software workshops.

In developing quality software products, NCR Self Service Systems Division has adopted the software development process (SDP) defined by the corporate engineering department. This process provides a uniform but flexible framework for planning, implementing and managing development programs to ensure timely, orderly and economic development of successful products. In the following sections NCR's current development process is described, followed by a detailed description of software test and documentation review procedures currently undertaken, with details of decisions on progression through development phases.

1.1 STANDARD DEVELOPMENT PROCESS

Software developed by NCR Dundee follows a corporate standard development process (SDP). The SDP organizes the development cycle into eight phases. A short description of each phase, the documents generated, review procedures and the requirements for progression to the next phase are set out below. Note that the following information has been extracted and summarised from an internal, confidential procedural document.

1.1.1 BUSINESS PLAN PHASE

The business plan phase is initiated when a development opportunity is proposed to product management for evaluation. Product managers within the product management department are responsible for individual products and product developments. They liaise with established and potential customers and the company's marketing groups to identify market requirements and business opportunities. This phase may also be initiated in response to requirements from the overall corporate product strategy. Based upon the decision by the appropriate management (local or corporate), product management issue a product requirements document (PR) for the approved development, containing functions/features required for the product and business objectives. This document authorizes the engineering developer to proceed with preliminary planning. The developer is a software engineer within the engineering department responsible for the design, development and implementation of the functions/features outlined in the PR document.

DOCUMENTS

A - PRODUCT REQUIREMENTS

This document is prepared by product management and initiates engineering activity on the development. The product requirements document may be continually updated, with interaction by product management and the developer

as the requirements are defined, but must be approved by the director of product management prior to termination of the business plan phase.

B - PROJECT PLAN SUMMARY

This document is prepared by the developer in response to a product requirements document and contains information on schedules, resources and funding required for the proposed project. It requires product management approval and is essential for termination of the business plan phase.

1.1.2 PLANNING AND DEFINITION PHASE

The purpose of this phase is to finalise the development planning activities and product specifications.

DOCUMENTS

A - PROJECT PLAN

This document provides a means for the developer to document the overall requirements necessary fully to describe and manage the development project. This plan consists of: a phase plan detailing the key phase dates of the development; a documentation plan outlining the documents that are required and the dates on which they should be available; a review plan specifying the departments responsible for reviewing and approving

documents produced; an organisation plan, highlighting the activities that are required of each department; a test plan detailing the formal tests required at each phase and the departments responsible for conducting the tests; a support plan, indicating the support activities required after product release; an education plan detailing the training requirements for development and support personnel, and customer education; and a risk analysis and contingency plan which highlights any potential risks that may cause delays in the project and proposed activities to alleviate schedule slippage. This document will be approved by the managers of the departments involved in the development.

B - FUNCTIONAL SPECIFICATION

This document provides a means for the developer to document the general concept, nature, purpose and intended functions/features of the product to be developed. For systems, a hierarchy of FS documents must be developed to specify the requirements of each subsystem or module. The document will be approved by the appropriate local management; for the system FS this is usually the general manager and local directors, for lower level FS documents approval is by the engineering manager and project leader.

The approval of both the PP and FS documents are deemed to terminate this phase.

1.1.3 DESIGN PHASE

The purpose of this phase is to define in detail how the requirements, specified in the functional specifications, are to be implemented.

DOCUMENTS

A - IMPLEMENTATION SPECIFICATION

This document is written by the developer for all systems, subsystems and modules for which a FS was written. The document provides design concepts, identifies subsystems, modules and their interfaces. The specification is approved by the responsible engineering manager and the plant maintainability and support (PM&S) manager. PM&S department will maintain the software product after release and their agreement on product implementation is required.

The phase terminates with approval of the implementation specification. Unit test specifications, that fully test each module, are also written during this phase. The documents are written by the developer and approved by the design quality assurance manager. Test specifications are not a mandatory corporate requirement during a development, but are included in "good practice" guidelines. Hence, approval of each test specification is not necessary for termination of this phase.

1.1.4 IMPLEMENTATION PHASE

The purpose of this phase is to complete software coding and conduct unit testing. Termination of this phase results from the completion and approval of unit test reports.

DOCUMENTS

A - UNIT TEST REPORT

A report is prepared for each software unit test performed and provides the developer with a means of documenting the results of that test; any exceptions or deviations from the specification during the test and corrective action undertaken to overcome these exceptions/deviations are reported. The report is prepared by the developer, with supporting information from a design quality assurance engineer, and is approved by the design QA manager.

It is the design quality assurance department's responsibility to approve the newly developed product prior to delivery to a customer. This includes ensuring that all phase end activities have been successfully carried out and that all tests, required to demonstrate the product's conformance to requirements, are performed during appropriate test phases. Integration and acceptance test specifications, that fully test subsystems and the product, are also written during this phase. The integration test specification is written by the developer

and approved by the design QA manager, the acceptance test specification is written by a design QA engineer and approved by the engineering development manager. Again, approval of each test specification is not necessary for termination of this phase.

1.1.5 INTEGRATION PHASE

This phase tests to show that lower level subsystems and modules, when integrated into systems, function together in accordance with the system functional specification requirements. Termination of this phase results from the approval of an integration test report and a design QA product release position statement.

DOCUMENTS

A - INTEGRATION TEST REPORT

A report is prepared for each software integration test performed and provides the developer with a means of documenting, as with the unit test, the results of that test. It is prepared by the developer, with supporting information from a design quality assurance engineer, and is approved by the design QA manager.

B - PRODUCT RELEASE POSITION

This statement quantifies the risks associated with

releasing the product, assessing the results of features/functions tested and initial review of customer level documentation. The statement is approved by the quality assurance director.

1.1.6 CERTIFICATION PHASE

The purpose of this phase is to verify that the product intended for delivery conforms to documented requirements and specifications and that the customer level documentation is complete. The acceptance test is conducted on the first production built hardware and demonstrates that the software operates properly in a simulated user environment. The approval, by the design QA manager, of an acceptance test report initiates the start of a customer verification test (CVT). A CVT will be carried out on all new product releases or major updates of existing products, at a selected customer site in a live environment. Termination of this phase results from product management issuing a customer verification test report. If a CVT is not required the phase will be terminated by the approval of the acceptance test report.

DOCUMENTS

A - ACCEPTANCE TEST REPORT

A report is prepared by a design QA engineer for the software acceptance test and provides the engineer with a

means for documenting the results of that test. This includes; a history of the problems raised during the test; hardware variants used; performance measurements; and competitive analysis measures, where applicable. The report is approved by the engineering manager responsible for the product.

B - CUSTOMER VERIFICATION TEST REPORT

This document is prepared by the CVT manager and contains statements of approval from the customer and local field personnel, involved in CVT. Position statements from both plant customer services and design quality engineering should also be included in this report.

1.1.7 FIELD FOLLOW PHASE

The purpose of this phase is that in the early installations at customer sites, necessary assistance will be provided to local support personnel to ensure successful installation and implementation of the product. There are no documentation requirements during this phase.

1.1.8 DISCONTINUATION PHASE

The purpose of this phase is to terminate all developer activities and transfer support responsibility to the

plant maintenance and support department. This involves archiving all records and documentation relating to the product development and producing a termination summary report. The completion of this report is deemed to terminate this phase of the development process and that of the program.

DOCUMENTS

A - TERMINATION SUMMARY REPORT

The report is compiled from a project history file, maintained throughout the development lifecycle. The report will contain; a complete record of the problems encountered and exceptions from the planned development; general comments on the development; and suggestions for improvements for future developments.

1.2 TEST/REVIEW PROCEDURES

On completion of the business plan phase, all development activities come under formal control procedures. During the planning/definition phase and design phase, the documents produced are validated to ensure they conform to the product requirements, that the requirements are complete and can be met. The documents produced during implementation, integration and acceptance phases verify, through testing, that the developed software meets the specified requirements.

1.2.1 INSPECTION PROCESS

Early in 1988 formal inspection techniques, for documentation validation and verification, were introduced.

Prior to formal inspections, informal review meetings were called where the document contents were discussed. Preparation time varied; expertise and experience of attendees varied, some participants attending not prepared to review the document but to learn more about the product the document was describing. Comments or changes were marked on the document, by the author, during the meeting but there was no guarantee that they would be included in the subsequent update. This review process was found to be inefficient with many reviews necessary before the document contents were agreed.

Formal inspections are now carried out on each document produced. This involves a formal meeting, of predefined duration, consisting of; the author, project leader and up to four peer group associates. The document to be reviewed is distributed to the review group members at least one week before the scheduled meeting, to allow adequate time for a detailed review. Standard inspection forms are provided to note any comments relating to the review document. These comments are noted during the preinspection review and consist of the query/error, the page and paragraph number and the type of error for each

comment. The error types are divided into nine categories, ranging from missing requirements to editorial standards. The time taken for this preinspection review is also noted on the inspection forms. The meeting is chaired by a moderator, usually the project leader, and a reader who is not the author will go through the document, summarising the contents. Comments are discussed and agreement reached on required changes as the reader describes each section. Any point raised during the meeting, not previously covered is noted by the moderator. At the conclusion of the meeting the moderator will decide if a reinspection is required. This decision is based on the number of non-trivial points raised. The moderator is responsible for ensuring that agreed changes are incorporated into the updated document and will enter a summary of results onto a database.

The database is used to record and produce statistics relating to the number of non-trivial problems per hour of review enabling comparisons of documents through the different phases and same documents for different products to be made.

Each piece of code also goes through a similar inspection process, fault noting and recording, the difference being the error type categories. The code is reviewed to ensure that it conforms to programming standards, that it is logically correct and covers all functional requirements.

1.2.2 SOFTWARE TESTING

Software testing is performed during implementation, integration and acceptance phases of the development process. A detailed description of each test type is given below.

Unit test

For each software module a unit test specification is written. The tests defined in the specification are of the 'black box' variety. Each function/feature of the software is tested with valid minimum and maximum input values, as well as invalid limits. The test specification is reviewed by a software design assurance engineer who also monitors the test. All problems found during unit test are recorded on an error reporting system. If problems are detected during this test phase a decision on whether to continue the test is taken, based on the nature of the problem. If the problem is such that it prevents a significant percentage of the module's functions to be exercised, the test will be abandoned, the problem fixed and the unit test rerun from the beginning.

If the problem affects one procedure or a small section of code the test will continue. When the problems have been corrected a decision is taken, based on the engineer's experience, on the amount of retesting necessary. The minimum amount will be to rerun the tests that failed. If a large number of minor problems were found, the above

tests plus tests that exercise the functions around the area where the problem was found, will be rerun. Where a large number of problems have been found, some of which are major, it may be necessary to rerun the complete unit test.

On completion of the unit test, a unit test report is generated that details the results of the test, a summary of problems found during the test phase and, where appropriate, a history of test reruns.

Integration test

When all modules have been through a unit test they are grouped together into a system, on test hardware, and an integration test is performed. This test has the same requirements as the unit test, in that all feature/functions are tested to a reviewed integration test specification, approved by the engineering and design quality assurance managers, and on successful completion of the test a test report is produced.

Acceptance test

The final inhouse test is the acceptance test. This again must be performed to a reviewed, approved test specification. The main difference between this test and the integration test, for small systems, is that it is performed on production hardware and is customer oriented. For large systems the integrated subsystems are brought together and the test is performed on the complete product. This test also verifies customer level

documentation.

Customer verification test

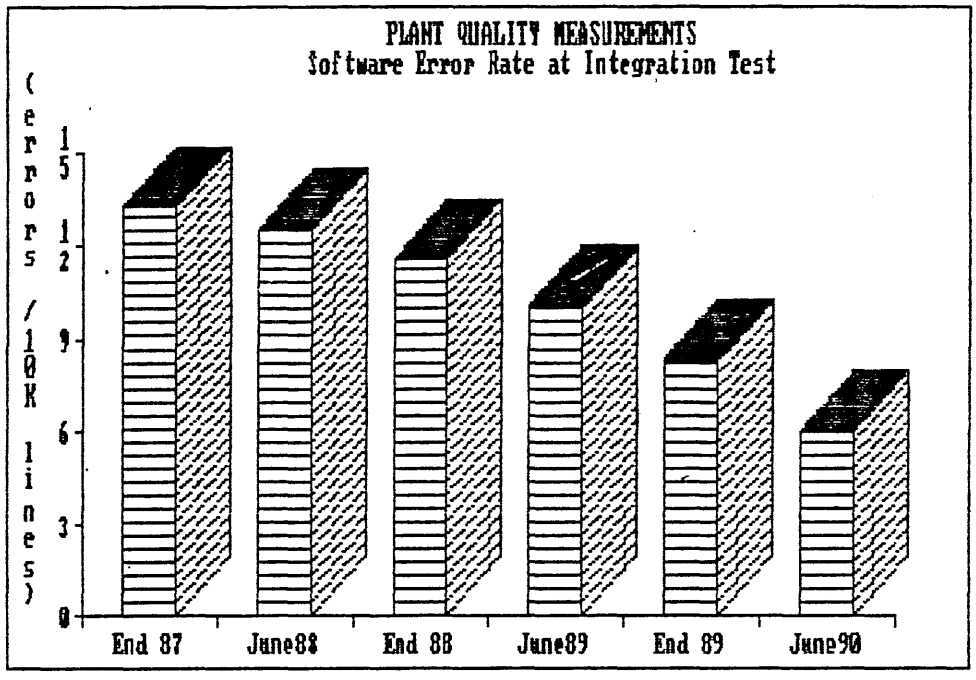
A C.V.T. test is performed on new hardware/software products, or major enhancements of existing products. This test is conducted by a customer, at the customer's premises, in a live environment. The test is controlled by product management and monitored by design quality assurance and field engineering. The purpose of this test is to verify all aspects of the product from training courses, customer service engineering expertise, customer level documentation, to the performance of the product in a live environment. On successful completion of this test the product will be made available for general customer usage.

CHAPTER 2 OBJECTIVES

One objective for all companies developing software should be to produce high quality software products. To determine if quality software is produced software quality metrics require to be formulated and measures taken. The results of these measures can then be compared with previous product developments, industry standards or even some predefined objectives. In the past the management of software developments and decisions taken during these developments have been based on subjective reasoning. Software metrics should form the basis for objective reasoning and decision making throughout the development of software products. If metrics are not collected consistently throughout a software development and over several developments the introduction of new automated development and test tools or modifications of the development process, for example, cannot be assessed.

This thesis will concentrate on software quality metrics and in particular examine software reliability models, to determine if the models can form a basis for progressing the software product through the various stages of the development process. The need to establish a software metrics programme has been recognised within NCR and in 1987 three basic measures were defined with longterm goals. These measures and the longterm goals are shown below, in graphical form.

Figure 2.1 - Software error rate at Integration Test

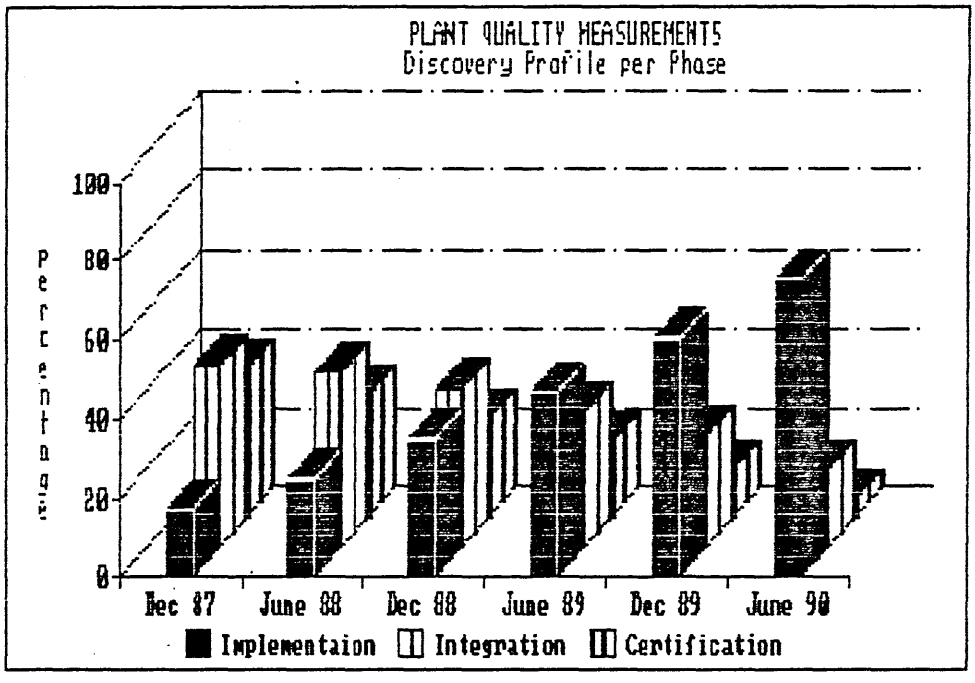


Dec 87	June88	Dec 88	June89	Dec 89	June90
13.23	12.50	11.50	10.00	8.30	6.00

Figure 2.1 shows the longterm goals for failures recorded during the integration test phase. This measure takes into account code size by normalising the error rate per 10,000 lines of executable source code. The figure for 1987 was obtained from failures recorded during integration test phase for all product developments at that time and obtaining an average value. It can be seen that the goal was to reduce the number of failures discovered during this test phase. With the introduction of new improved methodologies and development tools it was hoped, not only to reduce the overall number of failures discovered during a software development, but to push the discoveries back to earlier phases of the development. That goal is reflected in the measure shown in Figure 2.2

which is the failure discovery profile given as the percentage of failures discovered during three development phases. The first, implementation, records all failures discovered during design, code and module test phase: this includes all failures discovered during documentation inspections and code inspections as well as module tests. The second development phase, integration, records all failures discovered during integration test, with the final phase, certification, recording acceptance test found failures. The December 1987 figures again reflect the actual discovery profile, as an average, for the developments at that time.

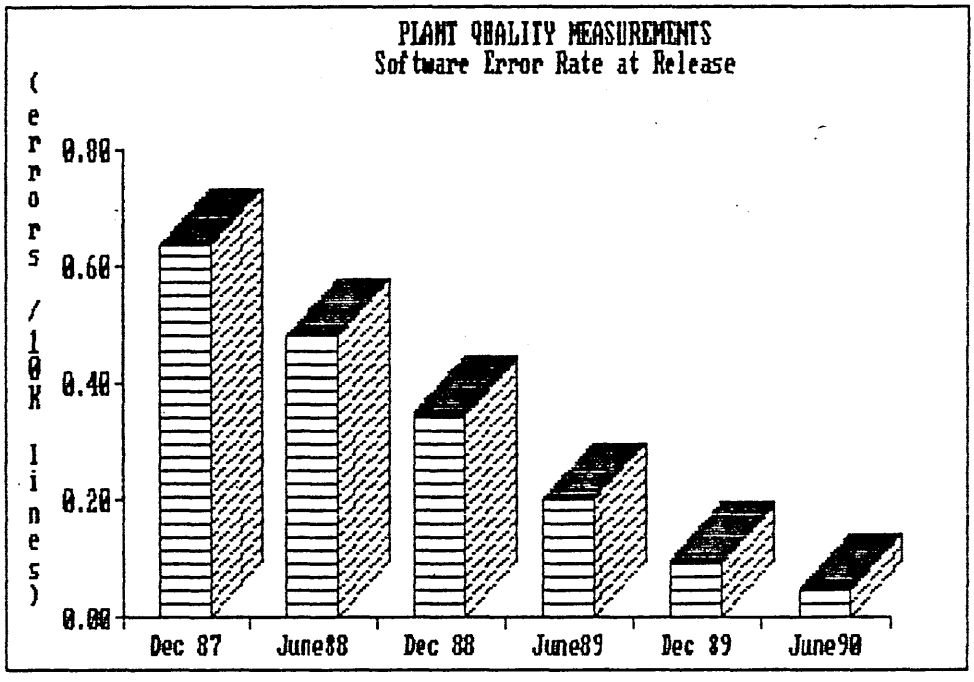
Figure 2.2 - Discovery Profile per Phase



	Dec 87	June88	Dec 88	June89	Dec 89	June90
Implementation	17.00	25.00	35.00	47.00	60.00	75.00
Integration	46.00	44.00	40.00	34.00	28.00	20.00
Certification	37.00	31.00	25.00	19.00	12.00	5.00

It is recognised at NCR that the development of error-free software for a product of any size is virtually impossible to achieve, as development costs and schedules have to be considered. The final measure, Figure 2.3, shows the goal for the failure rate at release (known errors remaining outstanding at time of release to customers), with the December 1987 figure again being an average of actual recorded failures for developments at that time. The 1987 guidelines for product release allowed for one error per 10,000 lines of executable source code to remain outstanding at release. The known errors must be minor in severity with acceptable workarounds. With the continual aim for improvement, this guideline has now been changed to one error per 20,000 lines of executable source code.

Figure 2.3 - Software error rate at Release



Dec 87	June88	Dec 88	June89	Dec 89	June90
0.63	0.48	0.35	0.20	0.10	0.05

While these metrics are useful for comparing product developments and assessing process changes, the decision making points in the development process, particularly the release decision, is still based on subjective reasoning and the development or quality managers' expertise and judgement.

Objective reasoning based on statistical techniques must be increasingly adopted in the decision making process. A measurement tool that looks promising is software reliability models. These models can be used to estimate the failure rate from a set of failure records, or perhaps more usefully, estimate the time to the discovery of the next failure. There is little point in releasing a product to customers if it is expected soon to become unsupportable. In the following chapter software reliability models are discussed in detail, including the estimation difficulties encountered.

CHAPTER 3 BACKGROUND TO SOFTWARE RELIABILITY

Reliability models for software systems began appearing in academic publications in the early seventies. Since then a large number of models have been suggested to describe the reliability of software systems. As stated by Stalhane [1] however, many of these models are basically the same model under different disguises. Mellor [2] suggests that all models can be classified into two distinct groups - Fault Manifestation and Interfailure Time. Fault Manifestation models assume that a software system contains a finite number of faults that cause failures at their own rate and independently of the other faults, whereas Interfailure Time models treat the time between failures as the random variable, with no regard to the total number of faults in the system. Two of the oldest, simplest and best known models, one from each group, are described below. It should be noted that the underlying assumptions of these models form the basis for most of the reliability models that have since been published.

The Jelinski and Moranda (J-M) model [3] is one of the earliest models from the so called Fault Manifestation group of models and has as a basic assumption that there is a finite and fixed number of faults in a software system with each fault equally likely to occur during software execution - faults are independent and failure times are exponentially distributed. The system has a

constant hazard rate (time between failures is negative exponential) until a fault is fixed, after which the hazard rate is again constant but with a smaller value - the hazard rate being reduced by a constant amount when each fault is fixed. A further assumption is that when a fault is found it is fixed immediately prior to the system being restarted and it is also fixed perfectly (no extra faults are introduced into the system).

The Littlewood and Verrall (L-V) model [4], from the Interfailure Time group, uses the concept of measuring reliability in terms of probability distributions for the time to next failure. As with the (J-M) model, however, it assumes continuous time and instantaneous repairs. More realistically, the model does not assume that the fix will be perfect and includes the possibility that the hazard rate may increase (the system may become less reliable) with time, by allowing the hazard rate also to have a probability distribution. As each fault is detected and removed the hazard rate should decrease because it is the programmer's intention to remove the fault, making the system more reliable. Experience at NCR has shown however that this is not always the case, with fixes not being correctly implemented. There have even been certain instances where the intended fix introduced an additional fault, or faults, into the system.

It can be seen then that the (L-V) model is conceptually more realistic than the (J-M) model. A later modification

to the (J-M) model by Miyamoto [5] however attempts to overcome the perfect fix assumption by adding an error introduction factor. There is another problem with the (L-V) model, which also affects the (J-M) model, in that it assumes that when a fault is detected it is immediately repaired. This instantaneous repair is not practical in a test environment but the problem may be overcome by not recording repeat failures due to known faults.

It was decided therefore to investigate the (L-V) model further, to determine if it could be used to estimate system reliability during software development. The model is described in greater detail in the following section.

3.1 THE LITTLEWOOD - VERRALL (L-V) MODEL

The model assumes that times to failure have an exponential distribution, i.e. if T_i is the running time between repair of the (i-1)th failure and the ith failure, then T_i has a probability density function (p.d.f.)

$$f(t,\lambda(i)) = \lambda(i)\exp(-\lambda(i)t) \qquad (t > 0)$$

The failure rate $\lambda(i)$ is a function of i, and the programmer's intention at the (i - 1)th fix is to make

$$\lambda(i) < \lambda(i-1)$$

However, this may not be achieved since some repairs introduce new bugs and make the program less reliable than it was before. So Littlewood and Verrall allow $\lambda(i)$ to have a probability distribution with the property that

$$P\{\lambda(i) < h\} \geq P\{\lambda(i-1) < h\} \qquad \forall\, i,h \qquad (1)$$

They propose a gamma distribution for $\lambda(i)$ since this gives two parameters for flexibility and also it is analytically tractable and has the correct range $(0,\infty)$. So the p.d.f. for $\lambda(i)$ in the (L-V) model is given by

$$g(\lambda,i,\alpha) = \frac{\{\psi(i)\}^{\alpha}\,\lambda^{\alpha-1}\,e^{-\psi(i)\lambda}}{\Gamma(\alpha)} \qquad \lambda \geq 0$$

where we have written λ for $\lambda(i)$, and the parameters of

the gamma distribution are α and $\psi(i)$. If we make $\psi(i)$ a monotonically increasing function of i, then it can be shown that condition (1) is satisfied.

Littlewood and Verrall proposed

$$\psi(i) = \exp(\beta_0 + \beta_1 i)$$

This certainly ensures that $\psi(i) > 0$ as is required for a gamma parameter, and it also ensures that $\psi(i)$ is an increasing function of i as required by (1). No other justification is offered for this choice of $\psi(i)$, and clearly many other functions could serve. Our choice of the exponential distribution for times between failures is based on experience with many kinds of failures; the choice of a gamma distribution for $\psi(i)$ is based on the considerations mentioned above; in setting $\psi(i)=\exp(\beta_0+\beta_1 i)$ we are making a much more arbitrary choice. Only experience will show if it was sensible.

However, Littlewood and Verrall used their model to generate 80 failure times with $\alpha = 2$, $\beta_0 = 2$, $\beta_1 = 0.2$. They then applied their model to the generated data, assuming α was known, and attempted to estimate β_0 and β_1. In practice α would not be known, but we start with the problem of estimating two parameters rather than three. As will be seen, even this is a formidable problem.

To test the goodness of fit of any reliability model we

can use the following method, which is an expansion of the description given by Littlewood and Verrall.

For any random variable U we can define its cumulative distribution function (c.d.f.)

$$F(u) = P(U \leq u).$$

Our probability model defines $F(u)$, and from it we can obtain $u(x)$ where $F(u(x)) = x$, for any real x.

Let $Y(x)$ be the proportion of observations $u_1,...u_n$ which do not exceed x. Then a plot of $Y(x)$ against x will be a step function with steps of $1/n$ occurring at points x_i given by $F(u(x_i)) = x_i$. If the model (and hence our values of x_i) is correct then this step function will fluctuate about a line of unit slope through the origin. Here we extend this to allow for the fact that each of our failure times T_i comes from a distribution with its own parameter $\psi(i)$.

So define $t_i(x)$ by the following:

$$P\{T_i < t_i(x)\} = x \qquad (2)$$

and hence a sequence of random variables $\{Y_i(x)\}$ by

$$Y_i(x) = 1 \quad \text{if} \quad t_i < t_i(x)$$

$$= 0 \quad \text{otherwise.}$$

and let

$$Y(x) = \frac{1}{n} \sum_{i=1}^{n} Y_i(x) \qquad (3)$$

$$= \text{proportion of } t_i\text{'s in the data which satisfy } t_i < t_i(x)$$

clearly Y(x) has expected value given by

$$E(Y(x)) = x \qquad \text{from (2).}$$

So if we can plot Y(x) against the ordered (ascending) x the result will be a step function. If our model, including our choice of $\Psi(i)$, is a good fit, it will fluctuate about a line through the origin with unit slope.

The steps in Y(x) are of magnitude 1/n if we observe failure times $t_1 \ldots t_n$, and they occur at points x given by solution of

$$t_i = t_i(x) \quad i = 1 \ldots n.$$

Littlewood and Verrall show that requesting closeness of Y(x) to the straight line is equivalent to specifying that the x_i are uniformly distributed on (0,1). Then a goodness-of-fit test can be applied.

It can be shown (Appendix A) that for the (L-V) model with their choice of $\Psi(i)$, we have

$$x_i = 1 - \left\{ \left(\log \prod_{m=1}^{i-1} k_m\right) \Big/ \left(\log \prod_{m=1}^{i} k_m\right) \right\}^{i}$$

where the k_m's are given by

$$k_m = \Psi(m) / (\Psi(m) + t_m)$$

So we can use a goodness-of-fit statistic applied to the plot of Y(x), or equivalently to the x_i's, as our criterion in a search algorithm to estimate β_0 and β_1.

3.2 ESTIMATION PROBLEMS FOR (L-V) MODEL

At the end of the previous section it was stated that a goodness-of-fit statistic is used as a criterion in the search algorithm. The first problem is in deciding which to use as there are several suitable goodness-of-fit statistics [6], of which Littlewood and Verrall used two, the nW^2 and the Kolmogorov-Smirnov. Both statistics are ways of deciding how close the step function is to the 45° slope. The Kolmogorov-Smirnov statistic is the simplest and is defined as the largest deviation from the 45° slope, whereas the nW^2 statistic is the sum of the deviations squared. Littlewood and Verrall gave no other details of their method of estimation for β_0 and β_1.

We used the Hooke-Jeaves [7] search algorithm and tried it with both nW^2 and the Kolmogorov-Smirnov statistic as criterion. The starting values for the first run were $\beta_0 = 2$, $\beta_1 = 0.2$. These are the true values, and in practice we can expect that our starting values will be very much worse than these.

The results were:

nW^2 : $(\beta_0, \beta_1) = (1.518, 0.20499)$

Kolmogorov-Smirnov: $(\beta_0, \beta_1) = (2.076, 0.20037)$

The results quoted by Littlewood and Verrall, without details of either the starting values or search algorithm, are:

nW^2 : $(\beta_0, \beta_1) = (1.5, 0.20640)$

Kolmogorov-Smirnov: $(\beta_0, \beta_1) = (2.3, 0.20085)$

Figure 3.1, at the end of this section, shows the value of the Kolmogorov-Smirnov statistic for $0.15 \leq \beta_1 \leq 0.25$ and $0 \leq \beta_0 \leq 10.0$. It can be seen that the minimum lies in a long narrow valley, somewhere between $\beta_1 = 0.15$ and $\beta_1 = 0.25$. Finding a minimum on such a surface is a difficult task for a search algorithm. When we changed the starting values, setting both β_0 and β_1 close to zero and using an initial step length of 2 for β_0, the program aborted with $\psi(i)$ becoming too large. The reason can be seen in Figure 3.2 . Once again there is a long narrow valley, but no minimum in the range $0 \leq \beta_1 \leq 2$. With starting values of 0.1 for both β_0 and β_1 and initial step size 0.05 , the estimates were $(\beta_0, \beta_1) = (1.556, 0.20499)$. The final step length for β_1 was 1.0e-10, and a plot of the Kolmogorov-Smirnov criterion appears in Figure 3.3 .

Clearly this estimation problem is non-trivial. Careful consideration must be given to the starting values and also the initial step-lengths. A large step length may jump the minimum, while a too-small step length will make the search unacceptably long. Even with a small step size, the search algorithm does not guarantee to find the minimum, possibly due to the presence of local minima.

The difficulties illustrated here are worse when nW^2 is used as the criterion in the search, and the final estimates are further from the true values as is seen in the results quoted from Littlewood and Verrall and from our run with starting values equal to the true values.

The problems of estimation in real life can only be worse, since we shall in practice have little idea of suitable starting values for β_0 and β_1 and also we shall not know the value for α, here assumed to be 2 throughout.

A major use of any reliability model is the prediction of future events. The decision on whether to release a software product should not be based solely on current perception of the product's reliability but also on when, in the future, the number of known errors will make the product unacceptable. This predicted time period should at least exceed the time required to produce a maintenance release.

We may ask how much poor estimates of β_0 and β_1 affect our estimates of reliability in the future. We can calculate

$$E(T_{n+1}) = \Psi(n+1)/(\alpha-1)$$

but the mean value of a markedly skewed distribution such as the exponential is very much influenced by high values which have small probabilities. Hence a more useful parameter to estimate would be the median time to the next failure, m, where

$$P(T_{n+1} < m) = 0.5$$

Littlewood and Verrall show that

$$m = \Psi(n+1)\left[\prod_{i=1}^{n} k_i\right]^{1-(1/(1-\alpha))}{}^{1/(n+1)} - \Psi(n+1)$$

where the k_i's are as defined above. If α is set to 0.5 we can calculate m, the median time to next failure. This value can now be used as T_{n+1} which enables us to estimate

a median time to next failure, T_{n+2}. This process can be repeated to estimate the total time from release to the time when an unacceptable number of known errors exist.

If the decision to release is taken after, say, 75 errors, with a further 5 errors being unacceptable we use the parameters estimated with the Kolmogorov-Smirnov statistic as a criterion, and then estimate the time between error 75 and error 80. The estimated median value for total time between error 75 and error 80 is given in the last column of the table, below.

Source	β_0	β_1	$T_{75}+T_{77}+..+T_{80}$
(L-V) paper [4]	2.3	0.20085	98,800,000
Our results using Hooke-Jeaves search	2.076	0.20037	92,020,000
Our result when an invalid minimum was found	1.556	0.20499	102,750,000
Actual parameters used to generate simulated data	2,0	0.2	88,870,000
Actual time generated in simulated data	-	-	67,750,000

From this it can be seen that all estimates were optimistic, with poor estimates for β_0 and β_1 giving excessively optimistic estimates for $T_{80}- T_{75}$.

The last column of the above table gives the estimated median values for each set of parameters. These values

however are the median of a random process and what would be interesting would be to compare the sampling distributions of the processes. The next section escribes the method adopted to simulate these processes, followed by a comparison of the distributions.

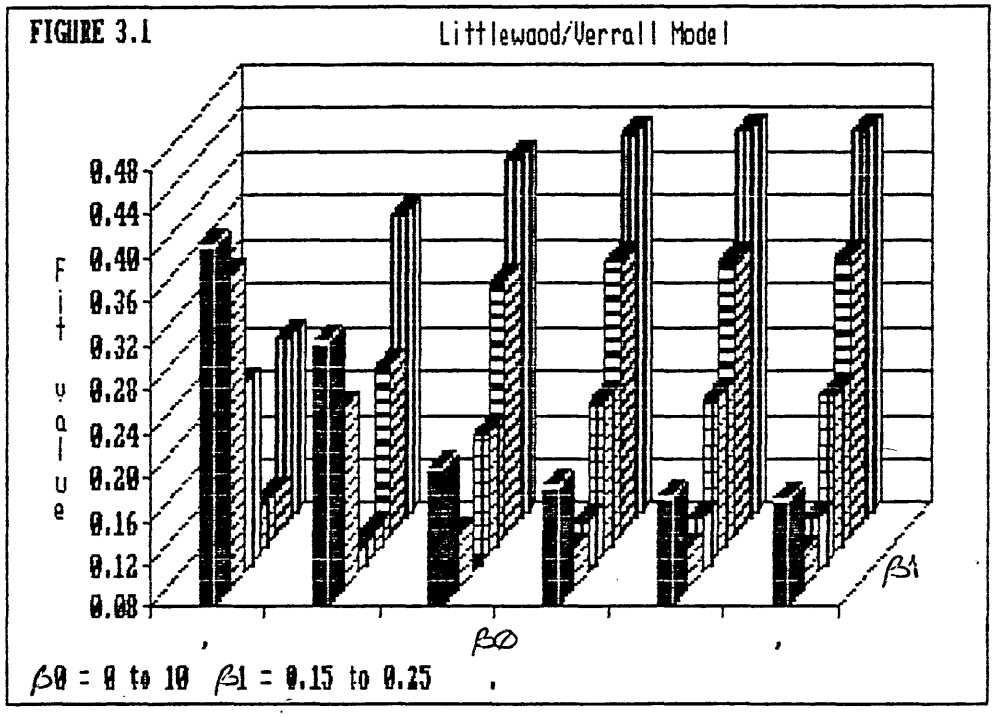
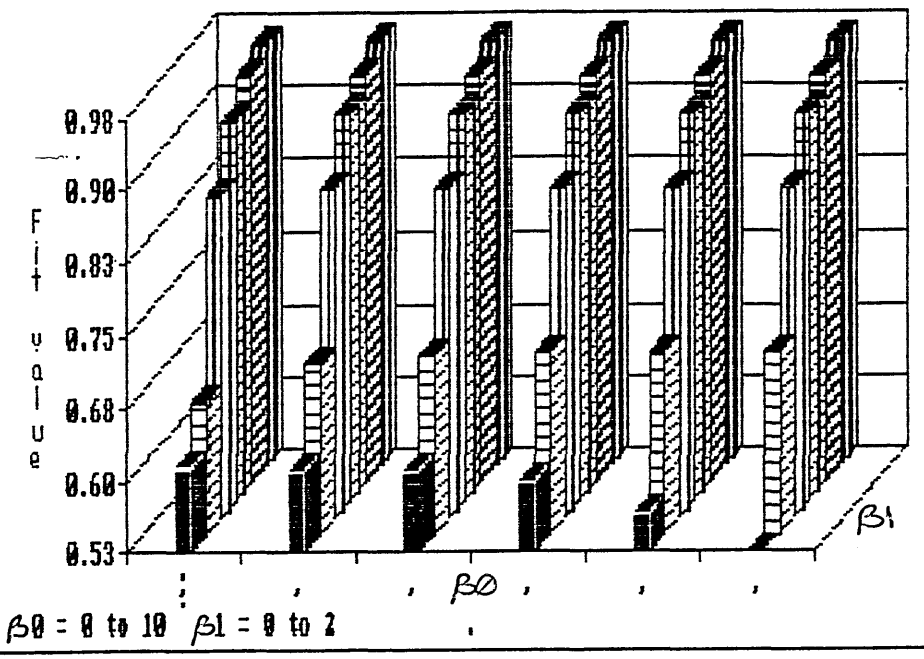
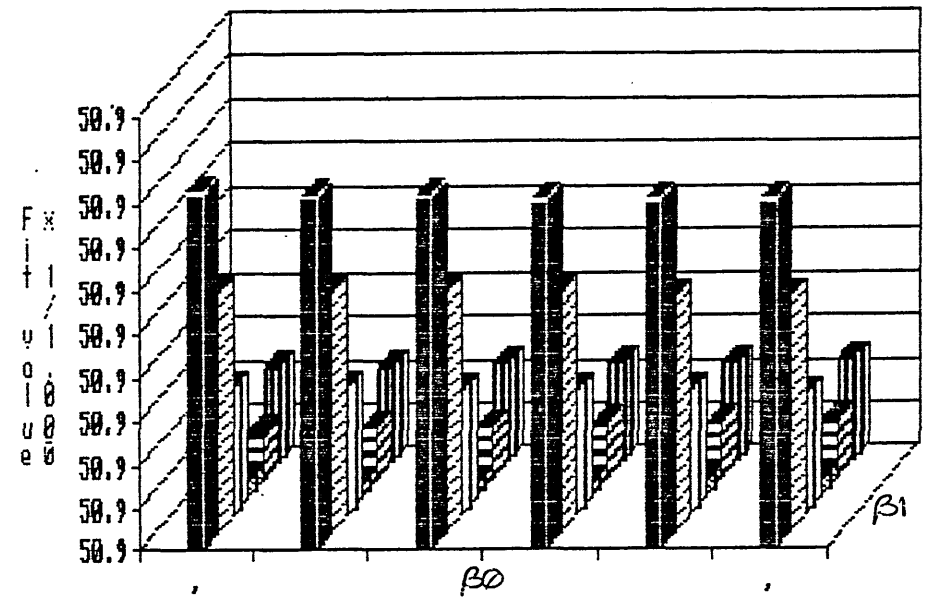


FIGURE 3.2 Littlewood/Verrall Model



β0 = 0 to 10 β1 = 0 to 2

FIGURE 3.3 Littlewood/Verrall Model



β0 = 1.5561864452 to 1.5561864457 β1 = .2049950273 to .2049950278

3.2.1 SAMPLING DISTRIBUTIONS BASED ON β_0, β_1 COMBINATIONS

To compare the estimated parameters for β_0 and β_1 combinations, contained in the table from section 3.2, sampling distributions were obtained for each β_0, β_1 combination. This procedure entailed generating random interfailure times for failure 76 through 80, adding these times to get total time from failure 75 to 80. This process was repeated 1000 times for each β_0, β_1 combination and histograms plotted, as shown below.

The random variate for each interfailure time was generated as follows. Three random numbers were generated, with the first two used to calculate

$$- \ln (\text{random1}) \; / \; \psi(i)$$

$$- \ln (\text{random2}) \; / \; \psi(i)$$

where $\psi(i) = \exp(\beta_0 + \beta_1 * i)$

this gives two random values from an exponential distribution. The result of these calculations are added together to obtain a random value for a Gamma distribution with a shape parameter of 2.

The final random number is then used in a similar manner as above to obtain a random value from an Exponential distribution, ie

$$- \ln (\text{random3}) \; / \; (\text{result from above})$$

Appendix C demonstrates validation of the random number generator used.

Figure 3.4

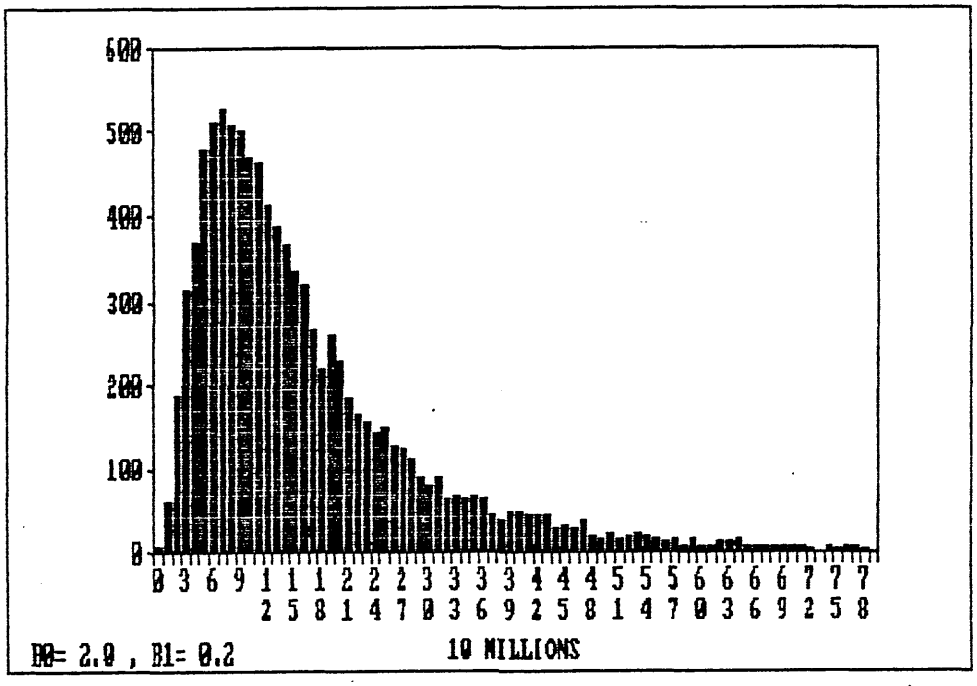


Figure 3.5

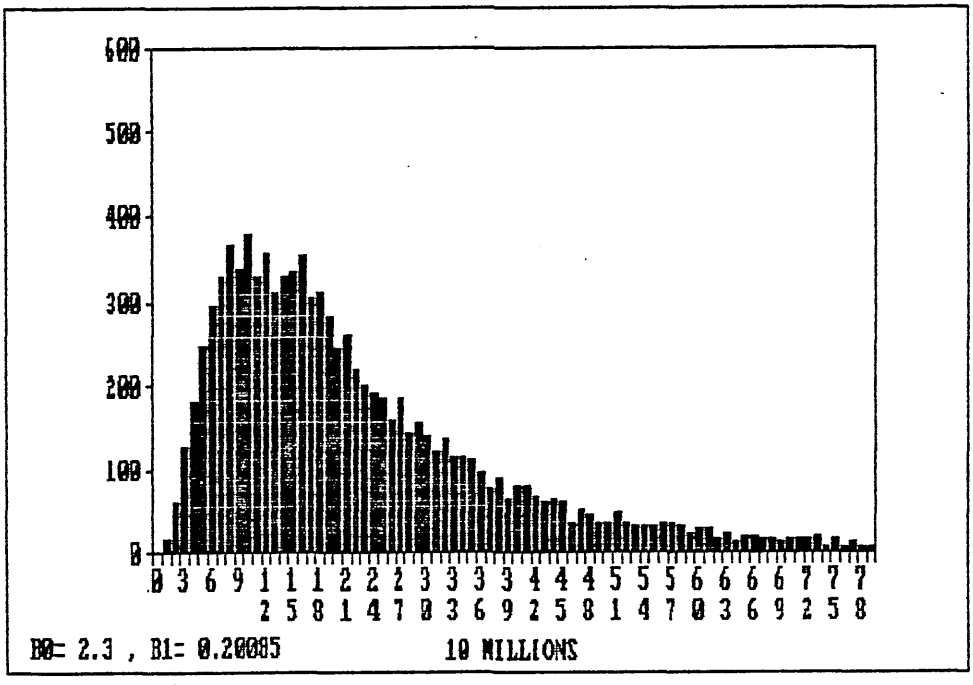


Figure 3.6

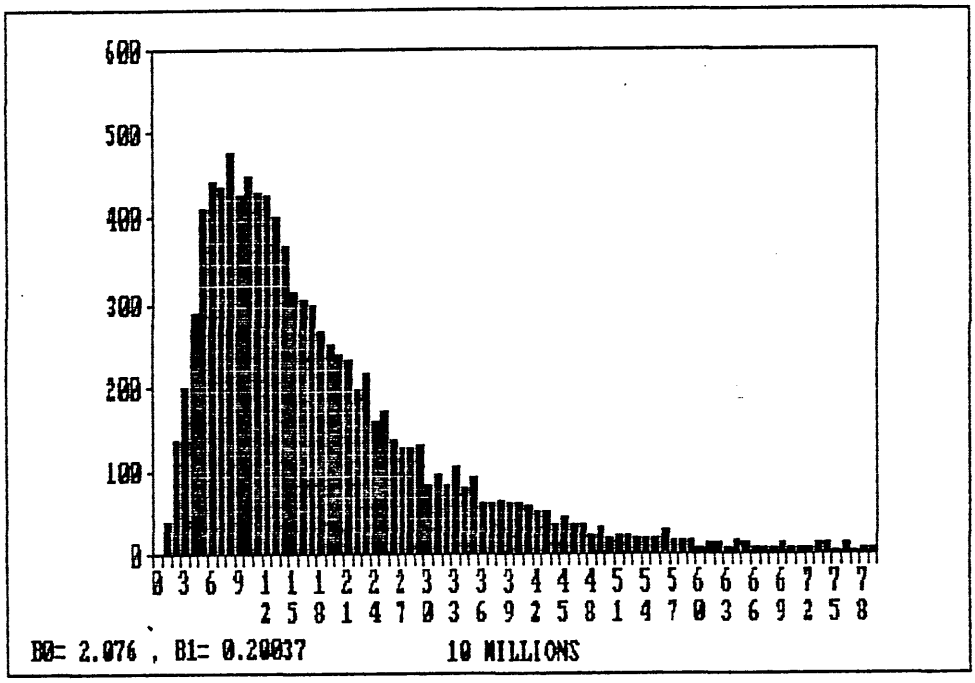
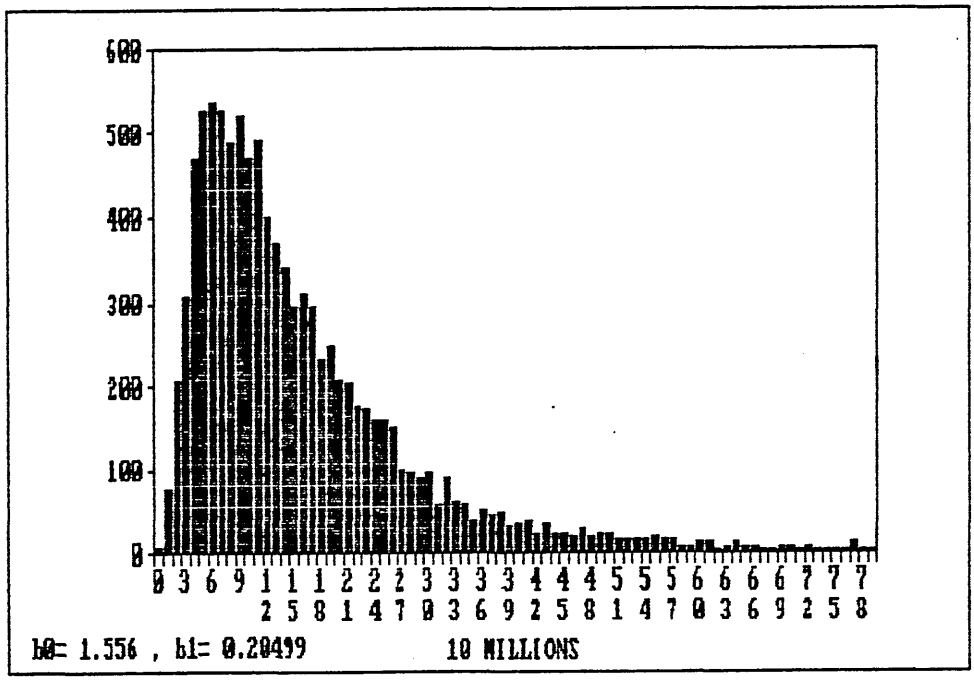


Figure 3.7



As can be seen from the histograms there are noticeable differences in the sampling distributions. How do these differences affect any decisions based on these distributions ? For example, given a set of estimated parameters, how long would it be before the software system contained an unacceptable number of known errors ? If a point on the histogram is taken such that 90% of the area under the curve lies to the right of this point it can be stated that we are 90% confident that the system will run for this time before it is unacceptable (at least 5 errors). Using the actual parameter values of $\beta_0 = 2$ and $\beta_1 = 0.2$ this point was found to be at approximately 41 million units. The parameters where a local minimum of the goodness-of-fit statistic was found ($\beta_0 = 1.556$ and $\beta_1 = 0.20499$) gave a pessimistic value of 38.5 million units. The value obtained using the estimate parameters found by the Hooke-Jeeves search algorithm ($\beta_0 = 2.076$ and $\beta_1 = 0.20037$) was approximately 48 million units. The final set of estimated parameters from the Littlewood and Verrall paper ($\beta_0 = 2.3$ and $\beta_1 = 0.20085$) in which they state that they considered these to be an acceptable estimate of the true values, suprisingly gave an overly optimistic value of approximately 62 million units. This value is 50% larger than the value obtained using the actual parameters. This does not correspond with the findings of Brockelhurst, S. et al. [8] who state that the Littlewood and Verrall model tends to give pessimistic predictions. If the predictions were used as a basis for a release decision there may be serious consequences for

the company in its ability to produce a timely maintenance release. The wide variations in predictions, given small fluctuations in the estimated parameters must raise serious doubts about the use of reliability models as a tool to aid the decision making process for software systems.

3.3 DEBUGGING MODELS

There have been a number of models published suggesting they can be used to measure software reliability during the test phase of software development, usually refered to as debugging models.

Scholz [9] also noted that many of the more sophisticated models are a special case of the Jelinski and Moranda model and that the models that relax the Jelinski and Moranda assumption of 'equal bug size' do not adequately address the statistical independence assumption. As discussed in section 3.0, one of the assumptions of the Jelinski and Moranda model is that the times between faults, or bugs, are independent and exponentially distributed. Scholz comments, correctly in the author's view, that during the debugging process faults are not independent of each other. For example, one fault may cause a failure that will not allow a large part of the software to be executed and this software may contain faults also. Some failure conditions may be the result of several faults that all require to be triggered prior to the failure manifestation. Scholz expands on the work carried out by Nagel et al. [10],[11] where the debugging process is carried out twice, the second run in a controlled manner. The debugging process is carried out correcting each fault that caused a failure, as it is discovered. This process continues until a certain number of faults have been discovered. The software is then

restored to its original form, faults being reinserted, and the debugging process repeated with random independent inputs until all known faults are rediscovered and a further fault found. It was stated that this information could then be used in the usual way to obtain an estimate for the residual failure rate of the system. Scholz found, using real data, that the estimates tended to be pessimistic and biased. Scholz did not discuss the possibility of finding previously undiscovered faults during the second random input run and how this would be incorporated into the model, nor does the model adequately address the situation where one fault masks other faults in the system. More importantly however, given the conclusion that this model produces biased results, there appears to be no economic justification for extending the test phase. The model proposed by Bittanti et al. [12] also attempted to relax the assumption of the Jelinski and Moranda model, that undetected faults are equally likely to occur, by changing the constant in the model to a linear function of the number of failures detected. While it is accepted that fault(s) discovery may be dependent on the removal of a particular fault, the introduction of a linear constant does not adequately overcome the independence assumption. In general the probability of detecting a fault in a certain piece of code will not be affected by fault(s) in totaly unrelated code within the system.

A further model that appeared promising was that proposed by Yamada, Ohtera and Narhisa [13]. They tackled the problem from a different angle by introducing testing-effort into the model. On closer examination however, it was found that the function for testing-effort was defined as either an exponential or Rayleigh curve, with no real justification. Does testing effort decrease as the software progresses through the development test phases and is the testing effort a smooth continuous curve? The answer to both of these questions is probably no, in the majority of software developments. As with the models previously discussed, this model has, as two of its basic assumptions, that faults cause random independent failures and that once discovered faults are immediately removed, with no new faults being introduced. As previously stated these assumptions are not valid for a software system during the test phase of development.

The final debugging model discussed in this section was developed by Raftery [14]. This is a Bayesian extension of the Jelinski and Moranda model and the same criticisms apply. The model assumptions do not reflect the test phase of software development. The estimates obtained, from these models usually tend to be optimistic and biased. And as concluded in section 3.2, there are difficulties involved in obtaining the model parameter estimates, with small variations leading to large variations in the estimates for time to next failure. The details of this model, therefore, will not be considered

further. What is of interest from this paper is not the ability to estimate time to next failure but a method of identifying whether the software under test is demonstrating reliability growth, as it progresses through the test phase. The mathematical justification for this reliability growth function is described below.

The model assumptions are;

System observed for period [0,T], during which n failures have occurred at times $t=(t_1, \ldots\ldots, t_n)$, where $n>1$.

Sample space consists of systems, therefore N (total number of faults) is a random variable.

N has a Poisson distribution, equivalent to a non-homogeneous Poisson process with rate function

M_1: $\lambda(s) = \rho \exp(-\beta s)$

where $\lambda(s)$ is the rate of occurrence of failures at time s

Testing for Reliability Growth

The paper compares the distribution M_1 with a constant rate Poisson process

M_0: $\lambda(s) = \mu$

The comparison of M_0 with M_1 is based on the Bayes Factor,

$$B_{01} = p(t|M_0)/p(t|M_1)$$

the ratio of marginal likelihoods.

Where, $p(t|M_0) = \int_0^{\infty} p(t|\mu,M_0)\,p(\mu|M_0)\,d\mu$

$$p(t|M_1) = \int_0^{\infty}\int_0^{\infty} p(t|\rho,\beta,M_1)\,p(\rho,\beta|M_1)\,d\rho d\beta$$

Raftery shows (see Appendix B for details) that the above functions can be redefined as ;

$$p(t|M_0) = C_0/T^n(n-1)!$$

and

$$p(t|M_1) = C_1(n-2)!/T^n \int_0^{\infty} y^{(n-1)}\exp(-Ry)/(1-\exp(-y))^{(n-1)}\,dy$$

Therefore B_{01} reduces to

$$(C_0/C_1)(n-1)\left[\int_0^{\infty}\exp(-Ry)\,(y/(1-\exp(-y))^{(n-1)}\,dy\right]^{-1}$$

The function can then be used on real data to determine if failure data is demonstrating reliability growth characteristics. Reliability growth is indicated if B_{01} is less than one. Raftery however quotes from the work by Jeffreys [15] who suggested that reliability growth should only be regarded as strong if $B_{01} < 0.1$ and decisive if $B_{01} < 0.01$. This technique will be used on real failure data, in chapter 5 .

This section has briefly described some of the published software reliability models, concentrating on one of the more promising, and attempted to highlight the estimation difficulties. These estimation difficulties resulted from the use of simulated failure data from a predefined

distribution. These difficulties will be worse when using real failure data and the problems compounded if incomplete or inaccurate information on failure data are collected. The following chapter describes potential problems in using these collected data in estimating the reliability of a software product.

4.0 DATA GATHERING

The use of software reliability models to estimate the reliability of software systems, or predict future reliability, requires that we record the time at which a failure occurs. Collecting these data is a difficult and time consuming task. If the data collection activity is not performed accurately and consistently the measures resulting from the application of these data to reliability models will be at best misleading and at worst meaningless.

NCR have recorded failure data and functional change requests for a number of years. These data are currently recorded on a central database, with on-line worldwide access, situated in the United States. The primary reason for maintaining this database is formally to record issues raised by customers, whether they are perceived faults or change requests, on NCR's complete product range and to track and monitor these issues to acceptable resolution. These issues, which are referred to as 'calls', are typically entered by the local NCR customer support personnel. However, divisions and individual plants within the NCR organisation also use this database to record calls on products during the development cycle prior to product release. This database, therefore, is the main source of failure data for NCR products.

The use of the database is as follows. When a call is first entered it is assigned a unique number and the date the call was entered recorded. The individual entering the call is then required to enter mandatory information and a description of the problem. The mandatory information consists of call type code that gives an indication of the type of failure encountered, a priority code reflecting the severity of the failure, a product identifying code and a responsibility code for the group responsible for investigating the call. The call will then be tracked by additional text sequences indicating information such as: problem investigation status, source of problem, details of any code change required to eliminate problem, verification details and an indication of the version of the product that contains a resolution of the call entry.

This database has been operational for the last three years. Prior to this it was the responsibility of the individual plants to record failure data relating to products under their responsibility. At NCR Self Service Systems Division, this was carried out on a local database which had been in operation for approximately five years. This local database was maintained for the same reason - to track a problem or functional change request through to resolution. This local database consisted of a current base and a number of history bases containing archived details of closed calls. Once the central database was fully operational the local base was scrapped. However,

to ensure these data were not lost and to allow the information to be used for comparison with current products, there was an attempt to transfer the information onto a personal computer. This turned out to be a lengthy and complicated process and it was found that some of the bases were corrupt, either through storing the bases on bad media or the media deteriorating. After the recovery activities and data transfer it was found that out of a total of 4658 record entries, 114 were lost. It is thought that this information can still be useful for comparison with current products and will be used in the following sections on data analysis and application of reliability models.

These databases have their limitations and any information extracted for use with reliability models should be used with care. As previously stated, these databases are used to track problems or change requests through to resolution with the information required for this activity being recorded. The current central database is aimed at problem tracking on released products and does not require information that would be useful for reliability analysis be entered, such as the software module involved, as the local customer support person would usually not be in a position to supply this information. Nor is the software development phase where the failure was first detected recorded - for released products this would tend to be delivery phase. The local database required these fields, module and development phase, to be recorded. Therefore

moving to the central database has meant that important information, for reliability prediction analysis, has been lost. While developers have been encouraged to record this information when entering calls on prereleased products, it has not been consistently entered. This means that the available information can only be automatically extracted on a product basis and any analysis of individual software modules would require manual searching which is open to errors and misinterpretation. Further obvious limitations in the use of these databases for reliability analysis is that when recording failures on released products no account is taken of the number of machines in operation by the particular customer nor when these machines were installed. Only the date the failure was recorded on the database is recorded and not the total execution time prior to failure. Also, during test phases, failures may not be recorded immediately they are discovered but collected and entered as a block at a convenient stage in the test.

In an attempt to overcome some of the limitations of data recording during the prerelease test phase, it has been decided to revert to a local database which will insist that the individual module is identified along with the development phase. It will also require that the total test time prior to failure discovery be recorded. While this will not overcome the possibility of recording inaccurate information it should improve the quality of

the recorded information. What it will not overcome is the individual's reluctance to record all failures found, particularly when testing software that they have developed personally, possibly for fear that it will be used as a measure of their competence and ability. This is thought to have occurred, and may still be occurring, so continued explanation of the reasons for this data collection must be given.

If the limitations of these data are taken into account, useful conclusions and decisions on the reliability of software products should still be possible. The data described in this section will be used in the following sections to analyse and measure the reliability of the software products developed by NCR Self Service Systems Division.

CHAPTER 5 DATA ANALYSIS AND ESTIMATION

The previous section described the mechanism used to record failure data and highlighted some of the limitations when attempting to apply reliability models to these data. This section will take into account these limitations and employ some exploratory data analysis techniques in an attempt to identify patterns in the data sets.

5.1 BASIC DATA ANALYSIS AND GRAPHICAL TECHNIQUES

The analysis will start by examining two recent software developments that resulted in the release of four software products. As previously mentioned, the current database for recording failure data does not insist on a development field entry. What can be identified however, through other sources, is the start dates for integration and acceptance test phases. The best disaggregation of phase information that can be achieved, therefore, is :

Implementation phase - all failures reported from the requirements phase through to unit test phase

Integration phase - failures reported during integration test

Certification phase - failures reported during acceptance test.

Three products released as a result of the first development consisted of an application, identified as APPLIC-1; a set of software tools to customise the application, identified as TOOLS-1; and a set of extension tools to allow the application to be modified and extended, identified as TOOLS-2. The second software development was an operating system, identified as OPSYS-1. The failure data from these developments are then compared with previous software developments, consisting of two applications, one operating system and a software tool set.

5.1.1 FAILURE DATA

The table below lists the total number of failures for each software product that resulted in a software change, broken down into the phase classification. The failures by phase are further subdivided into priorities, where priority 1 is a minor failure and a priority greater than 1 more serious (a rule for release being that a product can not be certified if it contains an outstanding failure with a priority greater than 1). The table also lists the number of lines of executable source code (LOESC) for each software product.

FAILURE DISTRIBUTION BY PHASE - recent releases

PRODUCT	**LOESC**	**IMPLEMENTATION**	**INTEGRATION**	**ACCEPTANCE**
OPSYS-1	142000	221	75	52
Priority (1,>1)		(137,84)	(40,35)	(30,22)
APPLIC-1	27000	32	9	46
Priority (1,>1)		(18,14)	(1,8)	(10,36)
TOOLS-1	102000	164	0	289
Priority (1,>1)		(83,81)	(0,0)	(128,161)
TOOLS-2	21100	22	0	11
Priority (1,>1)		(4,18)	(0,0)	(5,6)

It can be seen that only the first product in the table follows the expected trend of decreasing failures as the software product progresses through the development cycle. However even here, with approximately 15% of total failures on the product being recorded during the final test phase, it suggests that the earlier test phases are not sufficiently effective. The software tools (TOOLS-1 / TOOLS-2) did not undergo a formal integration test which is probably the reason for the large number of failures being recorded during the certification phase.

If these values are compared with the target plant measures for the appropriate time period, shown in the following figures, it can be seen that only the operating system (OPSYS-1) comes close to meeting the planned targets.

Figure 5.1

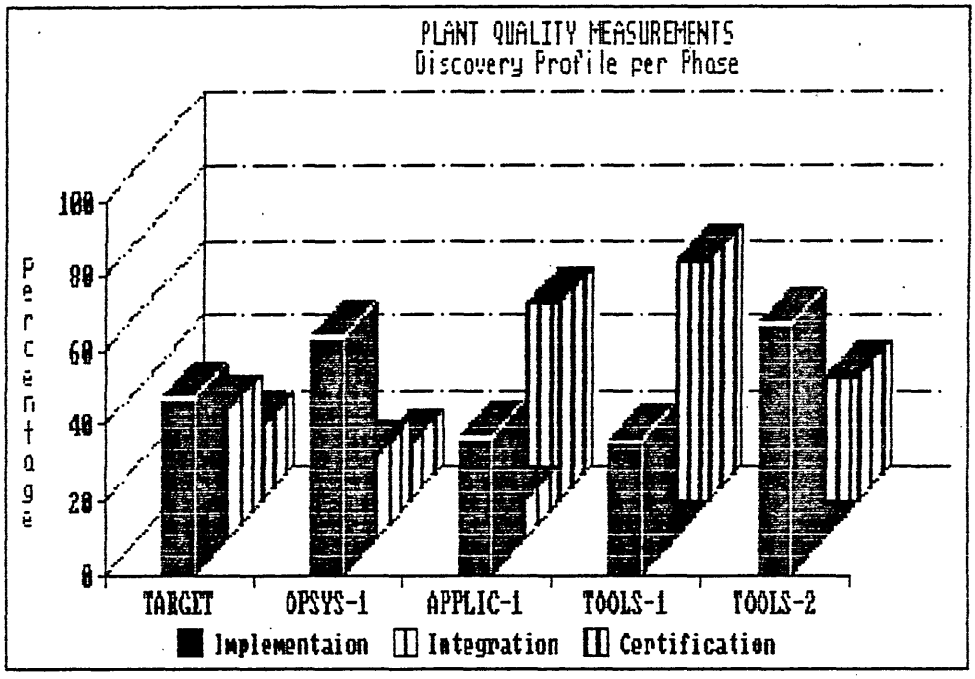
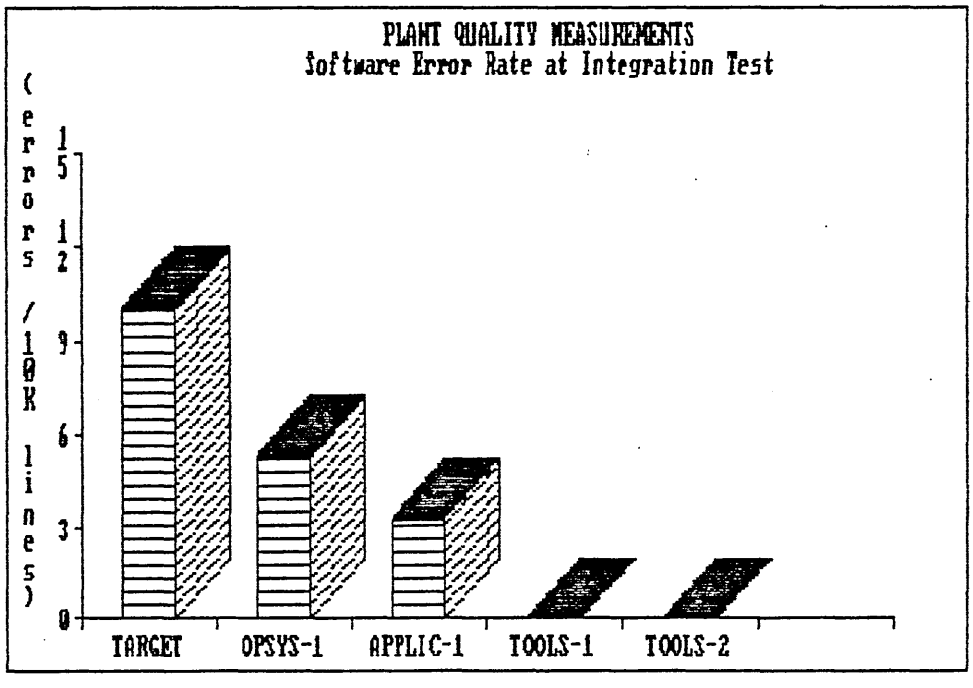


Figure 5.2



The table below lists total failures recorded for each product normalised by thousand lines of executable code. These failure density figures compare favourably with the results of six studies quoted in the work by Grady and Caswell [16]. Of the six projects quoted only two reported values of under 5, the remaining four varied from approximately 15 to 56.

FAILURE DENSITY - recent releases

PRODUCT ID	**OPSYS-1**	**APPLIC-1**	**TOOLS-1**	**TOOLS-2**
DENSITY/KLOEC	2.45	3.22	4.44	1.56

It is not suggested here that these figures show that NCR's software development process is vastly superior to the processes used in the reported studies. Neither is it suggested that because of the relatively small number of failures recorded, NCR have poor test processes. Failure densities can vary widely for many reasons, such as different development methodology, the varying use of development and test tools, different programming language, varying product size and complexity. They can also vary, as suggested by Grady and Caswell, because of differences in the definition of failures and how they are counted.

There are two possible explanations for the relatively low failure densities being recorded. The first is that the early test phases may not be rigorous enough, particularly with the application (APPLIC-1) and tools (TOOLS-1) products. This not only gives unusual failure

distributions but the large number of failures being detected later in the test phase, with increasing schedule pressures, increases the likelihood that not all errors will be discovered, which leads to a less reliable product being released to customers and a recorded failure density that is less than the true error value. The second reason, and probably most significant, is the under-recording of failures. The module and integration test phases are conducted by the individual, or group, who developed the software and while this test activity is monitored by an independent group it is possible that a failure may be discovered and fixed without being recorded on the error reporting system. It is also possible that debugging activity and test 'dry runs' will be performed before the official test and any failures discovered during these activities may not be recorded. Individuals are always reluctant to report failures against their work for fear that these figures may be used as a measure of their ability or productivity.

For these reasons, the failure densities reported above may be less than the actual values for these products. However, as the development methodology and failure reporting techniques have not changed significantly in recent years, useful information may be obtained by comparing these gross measures with measures from previous software developments. The table below lists failures recorded on a further four software developments. The first two products are developments from 1984, an

operating system (OPSYS-2) and an application (APPLIC-2). The other two products, an application (APPLIC-3) and a set of software tools (TOOLS-3), are developments from 1987.

FAILURE DISTRIBUTION BY PHASE - previous releases

PRODUCT	**LOESC**	**IMPLEMENTATION**	**INTEGRATION**	**ACCEPTANCE**
OPSYS-2	66000	214	36	135
Priority (1,>1)		(206,8)	(31,5)	(86,49)
APPLIC-2	22000	5	11	77
Priority (1,>1)		(2,3)	(10,1)	(43,34)
APPLIC-3	29700	45	162	35
Priority (1,>1)		(39,6)	(110,52)	(19,16)
TOOLS-3	11600	0	56	101
Priority (1,>1)		(0,0)	(43,13)	(55,46)

This gives failure density figures for the above products

FAILURE DENSITY - previous releases

PRODUCT ID	**OPSYS-2**	**APPLIC-2**	**APPLIC-3**	**TOOLS-3**
DENSITY/KLOEC	5.83	4.23	8.15	13.53

From these figures it would appear that the failure density measures rose significantly in 1987 from the 1984 figures, then dropped off to below the 1984 values during 1989. However care should be taken when comparing failure density measures. There are three types of software products under review - operating system software, application software and software tools, each with their own characteristics. It may not be appropriate to compare

an operating system development with a tools development, for example. The recent software developments involve much larger and more complex software products and while the normalising factor of lines of code should take into account size differences, can the complexity of the products be taken as equal?

The problem is further compounded by the fact that for the recent software development a tool was used to calculate executable lines of code, for previous developments an estimate was used. This estimate was obtained by counting the number of lines from a sample of source code and compiling this sample to determine the number of bytes in object code. The object code size was obtained for the product as a whole and an estimate for total lines of source extrapolated from these figures. While it is expected that the values were a good approximation, there is the possibility that these measures may have been inaccurate.

If the software developments are compared, it can be seen that the operating system developments recorded the same order of failures (OPSYS-1 - 348, OPSYS-2 - 385), however in the recent development there is a smaller percentage of failures discovered during the final development test phase (14.94% for OPSYS-1 as apposed to 35.06% for OPSYS-2) and coupled with the fact that this development was significantly larger (OPSYS-1 - 142KLOC , OPSYS-2 - 66KLOC), it shows an encouraging trend. This is not true

of the other developments, with only the application development from 1987 (APPLIC-3) showing a relatively small percentage (14.46%) of problems recorded at the final test phase.

It had been recognised that too many failures were being recorded at the final test phase, whether because of nonrecording in earlier phases or poor testing. This was particularly true of application integration testing. During the test phase for application APPLIC-3 therefore, a big effort was made to improve the integration test phase and record all failures on the error reporting base. The improvements can be seen, as approximately 67% of total failures recorded on this product were recorded during the integration phase, with only 15% recorded during acceptance test. Unfortunately this effort was not maintained through later developments.

The only test phase where failures are consistently recorded is during the final acceptance test. Comparing the failure density measures of failures reported during acceptance tests will remove the uncertainty of under-recording during the early test phases but will not overcome problems associated with different product types, complexity issues and possible inaccuracies in lines of code measures. The measure is given in the table below, where the percentage of total failures recorded during acceptance test is also noted.

FAILURE DENSITY for failures recorded during accpt. test

PRODUCT ID	**OPSYS-1**	**APPLIC-1**	**TOOLS-1**	**TOOLS-2**
DENSITY/KLOEC	0.37	1.70	2.83	0.52
ACCPT. TEST %	14.94	52.87	63.80	33.33
PRODUCT ID	**OPSYS-2**	**APPLIC-2**	**APPLIC-3**	**TOOLS-3**
DENSITY/KLOEC	2.05	3.50	1.18	8.71
ACCPT. TEST %	35.06	82.80	14.46	64.33

If we look first at the operating systems, it can be seen that not only does the failure density reduce, from OPSYS-2 to OPSYS-1, but the percentage of failures recorded during the acceptance phase also reduces. This reflects overall improvements in the development process, a more comprehensive integration test phase and better recording at earlier phases.

If the application developments are considered, there is an overall reduction in the failure density from the 1984 development, APPLIC-2 to the 1989 development APPLIC-1. There is also a reduction in the percentage of failures recorded during acceptance, however with over 50% of total failures still being recorded at this test phase there is still considerable scope for improvement, mainly in the area of consistent failure reporting. This is emphasised by the figures for the 1987 application development, where a comprehensive integration test was performed with all failures being recorded, which shows the smallest failure density figure and just 14% of total failures recorded during the acceptance test phase.

Finally, comparing the tools developments, the percentage of total failures recorded during the acceptance tests are the same but there is a significant reduction in failure densities. Looking back at the two acceptance test phases however, the quality of the latest software tools product at acceptance test did not appear to be any better than the previous development. In fact both products missed their respective release dates and required extra resources to complete the acceptance test phase. One possible reason for the difference in failure density measures is that the first software source line count was an estimate and may have underestimated the actual figure. However, it was expected that the most recent product would have a lower failure density because, even given that this product was considerably more complex, experience gained from the initial development through design and test methodologies would have helped improve the quality of subsequent, similar projects.

While improvements in the development process and experience gained from previous software developments have been reflected in a lowering of the overall failure densities, there are still too many failures being recorded late in the development/test cycle. This point is reinforced in the following section where graphs of the cumulative failure profile, for the product developments under review, are shown.

5.1.2 CUMULATIVE FAILURE PROFILE

The following plots show the software failure discovery profile for the development lifecycle. These detail the cumulative failures plotted against time. Both axes are normalised. Product size is taken into consideration with cumulative failures per 20,000 lines of executable source code, plotted against time and the time axis normalised over the reporting period. The plots show total recorded failures and these failures subdivided into priorities, where the priorities are as described above.

Figure 5.3

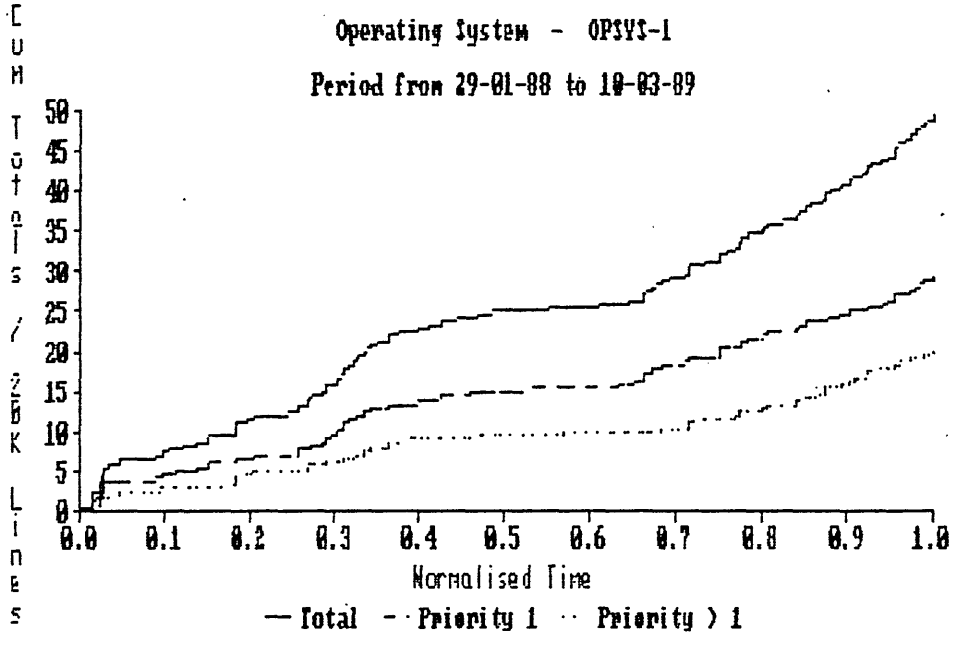
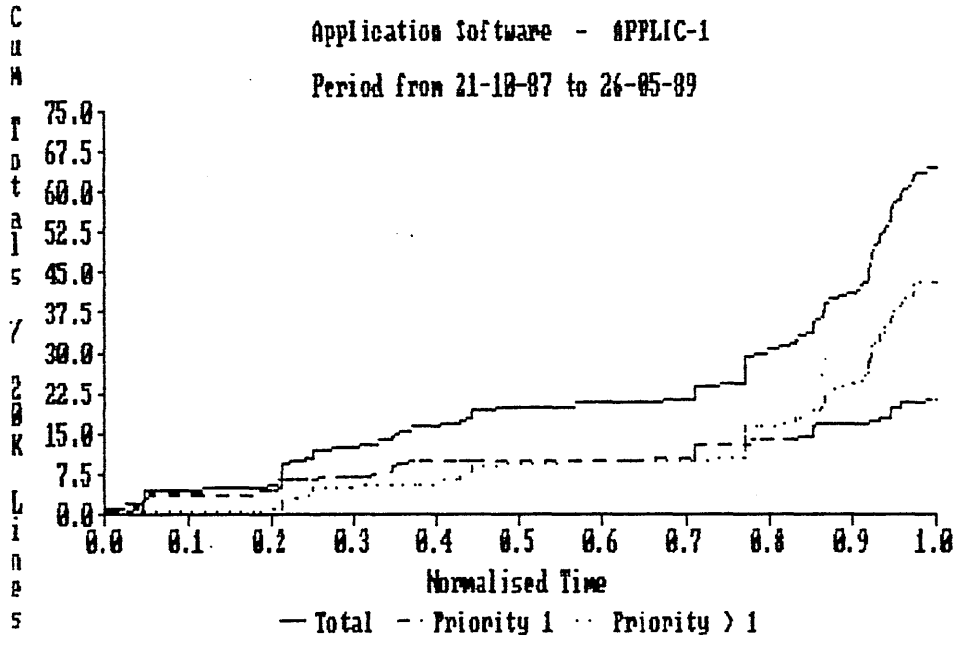


Figure 5.3 shows failures reported on the recent operating system OPSYS-1, from the start of the development through to product release. The first half of the graph shows the cumulative failure plot rising in steps, indicating

failures reported during individual module unit tests, which are spread throughout the early development phase. The cumulative plot levels off prior to integration test then rises steadily from the beginning of integration test to the end of acceptance test. This steady rise through continuous integration/acceptance testing unfortunately shows no indication of levelling off (which would indicate increasing time between recorded failures) and therefore questions the justification of the decision to release.

Figure 5.4

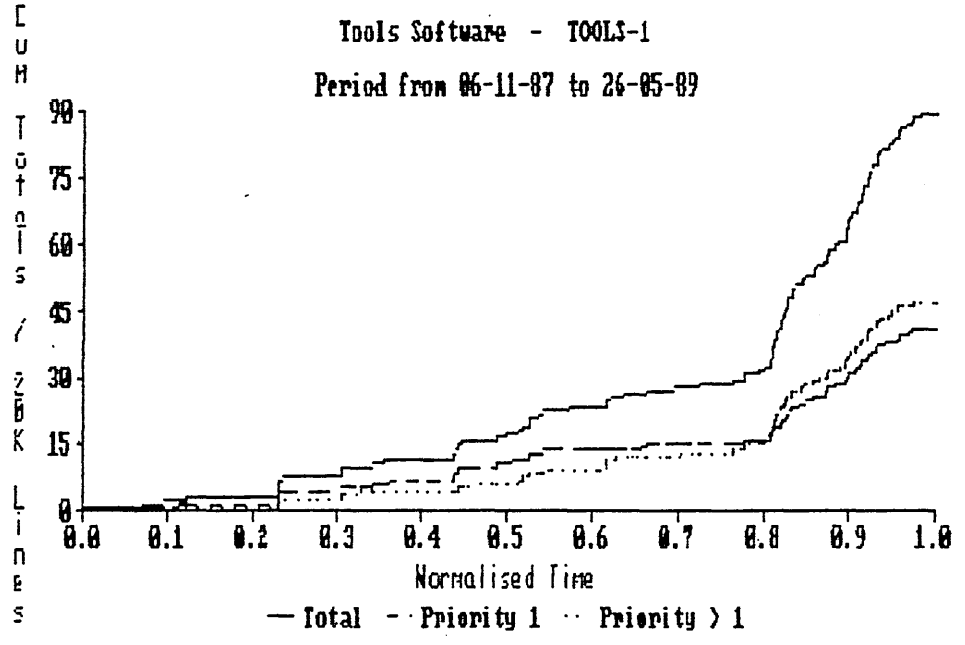


The second graph Figure 5.4, plots failures recorded over the development of the recent application product APPLIC-1 and clearly shows a sharp increase in failures reported over the last quarter of the development. The last quarter of the graph shows failures recorded during the acceptance test phase and the two distinct curves within this period can be explained as follows. The software tools must first be used to produce output to configure the application and during the first phase of acceptance test most of the test resource was applied to the tools and only in the final phase was the application fully validated. The final phase however, does show the cumulative profile levelling off, indicating increasing time between failure reports.

If we now consider failures recorded over the development

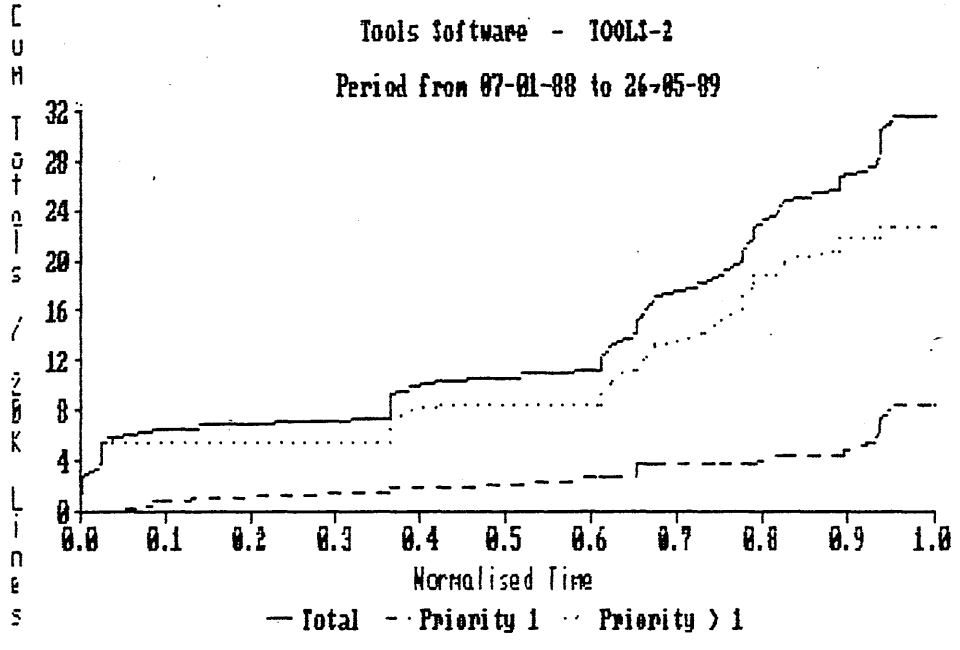
of the recent tools product TOOLS-1, Figure 5.5 below clearly shows a sharp increase in failures reported over the last fifth of the development. This portion of the graph reflects failures recorded during the acceptance test phase and as with the application product there appear to be two distinct curves. However the reasons are not the same: during the acceptance phase updated software was regularly released to the acceptance test group and after the large number of problems recorded at the start of acceptance test, indicated by the initial steep rise in the cumulative curve, resources were utilized in regression testing to confirm the correct implementation of the software fixes and therefore new problems were not being recorded at the same rate. However the final phase, as with the application product, does show the cumulative profile levelling off, again indicating increasing time between failure reports.

Figure 5.5



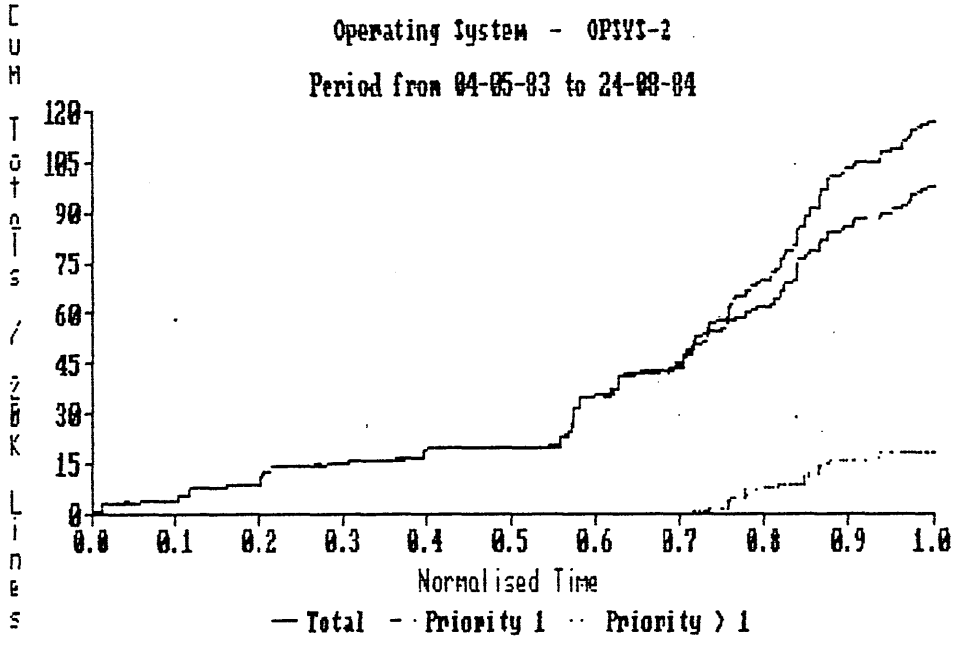
The final graph for recent products, Figure 5.6, shows failures reported on the tools product (TOOLS-2). As can be seen, the cumulative curve is similar to that of the tools product, showing a steep increase in failures reported during the final phase of development. While the shape is similar, the overall failure figures are better, with just under 32 failures per 20K lines of executable code, compared with nearly 90 for the tools product TOOLS-1.

Figure 5.6



The following graphs plot the cumulative failure profile for the earlier software product releases and all show a levelling off towards the end of the development phase, indicating an increased time between failure reports. The first, Figure 5.7, shows the earlier operating system development (OPSYS-2) and the sharp increase again indicates the start of acceptance test.

Figure 5.7



The next graph, Figure 5.8, shows the application development (APPLIC-2) and the time period is from the start of acceptance test. This reflects both the poor standard of earlier test phases and the nonrecording problem. The fact that 16 problems were recorded against earlier test phases, reported on table in section 3.1.1, during this period can be explained by developers' recording failures in updated software, undergoing regression test prior to release to the acceptance test group.

Figure 5.8

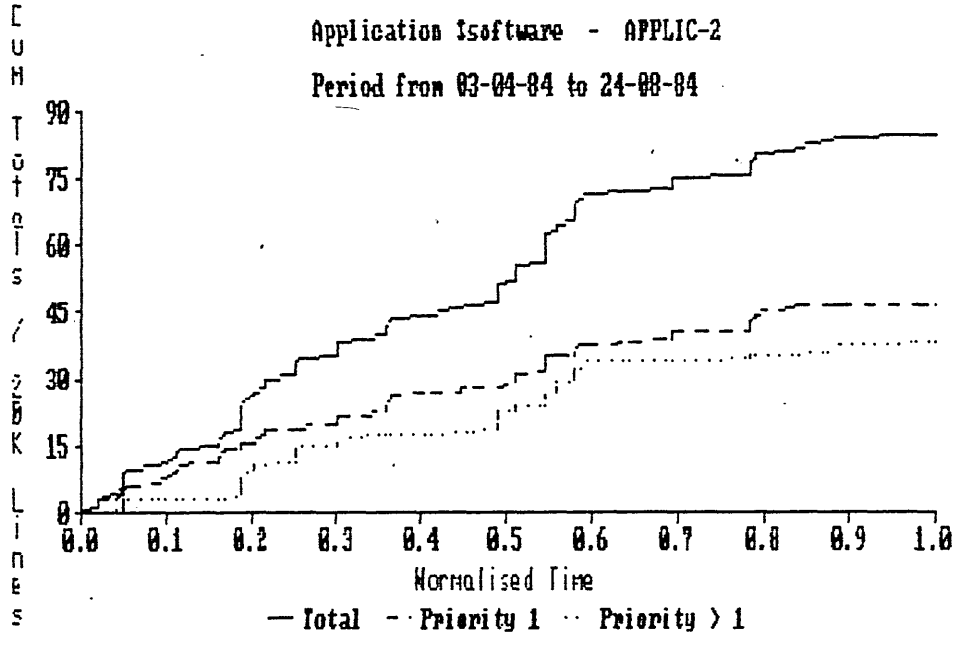
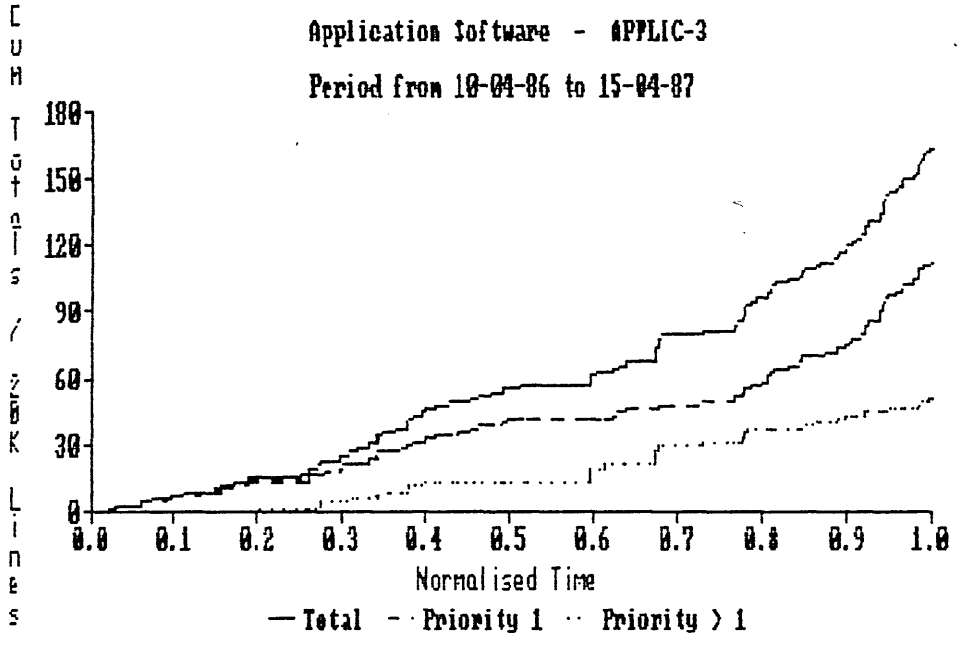


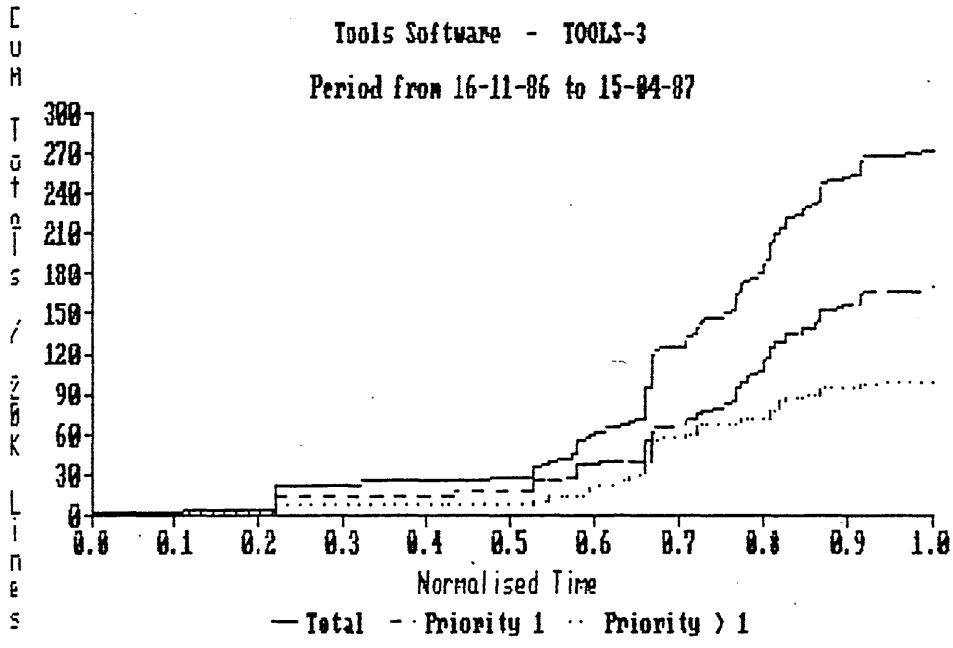
Figure 5.9 shows the 1987 application development. This was the development where a determined effort was applied to the integration test phase, with a concentrated test effort ensuring that all discovered failures were logged on the failure database. This is reflected in the cumulative curve which rises earlier in the development cycle and less steeply during the acceptance test phase than the other software developments. However, it does not show any indication of levelling off at the end of the development.

Figure 5.9



The final graph, Figure 5.10, shows the 1987 software tools development (TOOLS-3), and with the curve rising steeply at the end of the development phase this again indicates the poor standard of earlier test phases and the nonrecording problem.

Figure 5.10



5.1.3 TIME BETWEEN FAILURES

This section will examine the time between reported failures for the recent products under review. As discussed in chapter 4, the recorded time of failure is the date the failure was recorded on the reporting system and not when the failure was actually discovered. This results in a large number of record entries of the same date. For analysis purposes time will be divided into hours, with the number of failures recorded on the same date divided by 24 to give a TBF in hours. During test phases, particularly acceptance test when the scheduled certification date looms ever closer, the test activity is performed seven days a week; however failures discovered over the weekend will not be recorded on the database until the following Monday. The data therefore have been further manipulated by dividing the failures recorded on a Monday by 72, to reflect this weekend test activity. The reason for this is shown in the following two graphs that plot the failures reported during the first five weeks of acceptance test of the recent tools development TOOLS-1. As can be seen from the first graph, Figure 5.11 recording by days, there are five pronounced steps in the data. Four of these steps, at approximately 0.2 intervals, reflect failures recorded on a Monday. The second graph, Figure 5.12 shows the same test period with the failure data recorded as hours, manipulated as described above. It can be seen that the failure profile by hours is smoothed out, hopefully more accurately reflecting failure

discovery, rather than failure reporting.

Figure 5.11 - Normalised time in DAYS

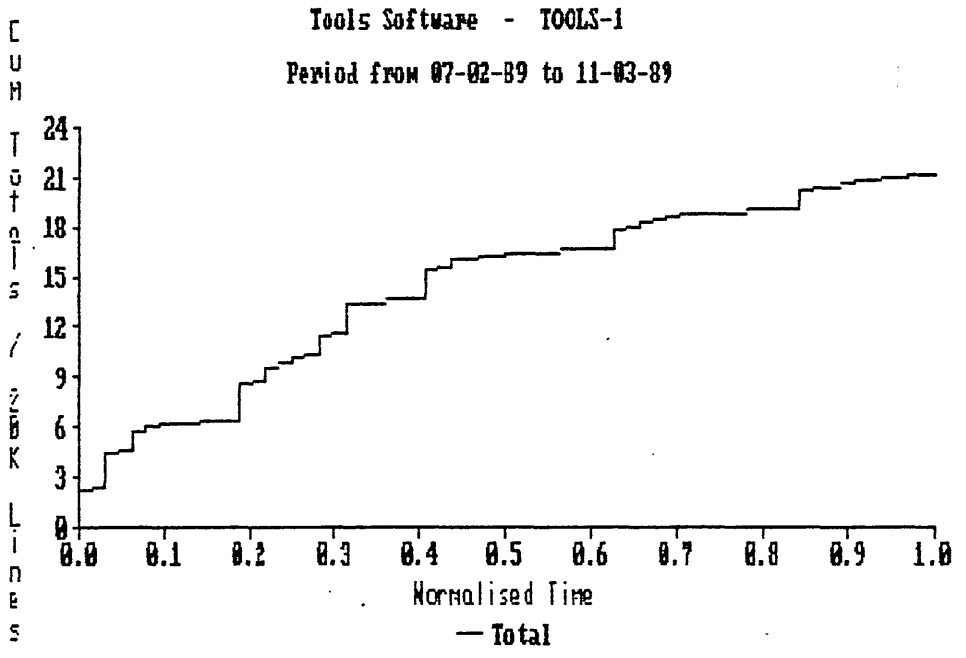
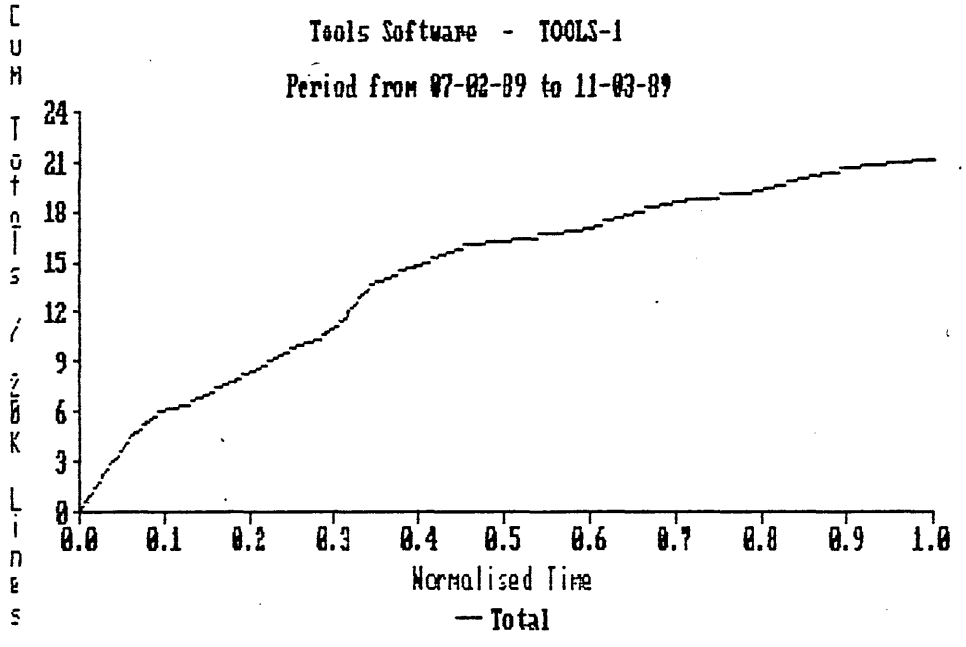
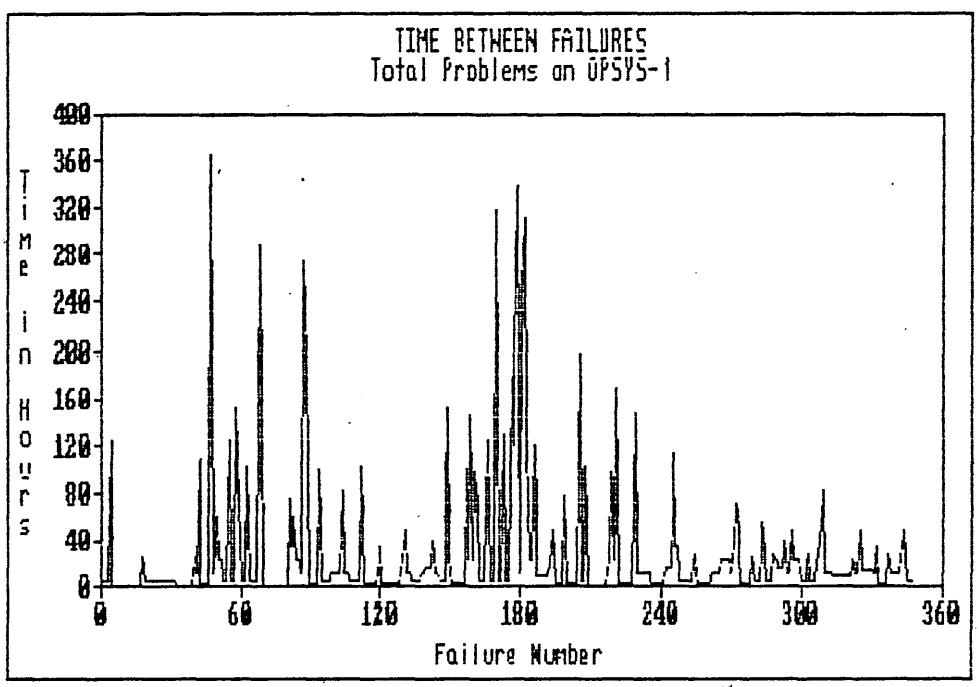


Figure 5.12 - Normalised time in HOURS



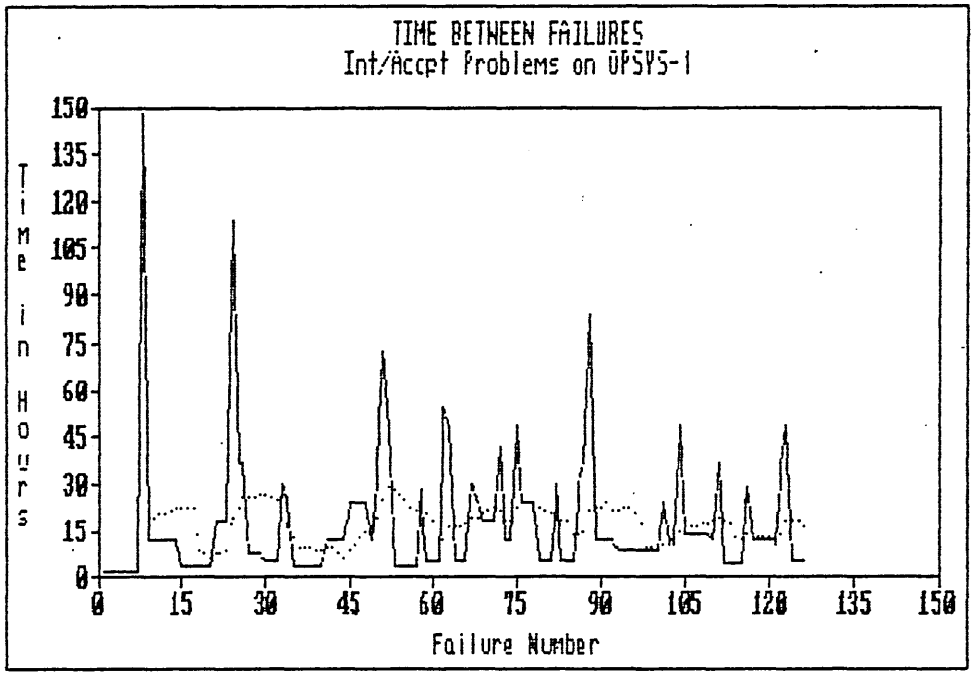
The following graphs plot the time between failures for the recent software developments. If we examine the operating system product (OPSYS-1) first in Figure 5.13, it can be seen that there is no pattern of increased time between failures for the development as a whole. Up to approximately failure 180 there is a cyclical trend, highlighting the discrete nature of unit test activities. From this point through to release the trend appears to be one of decreasing time between failures.

Figure 5.13



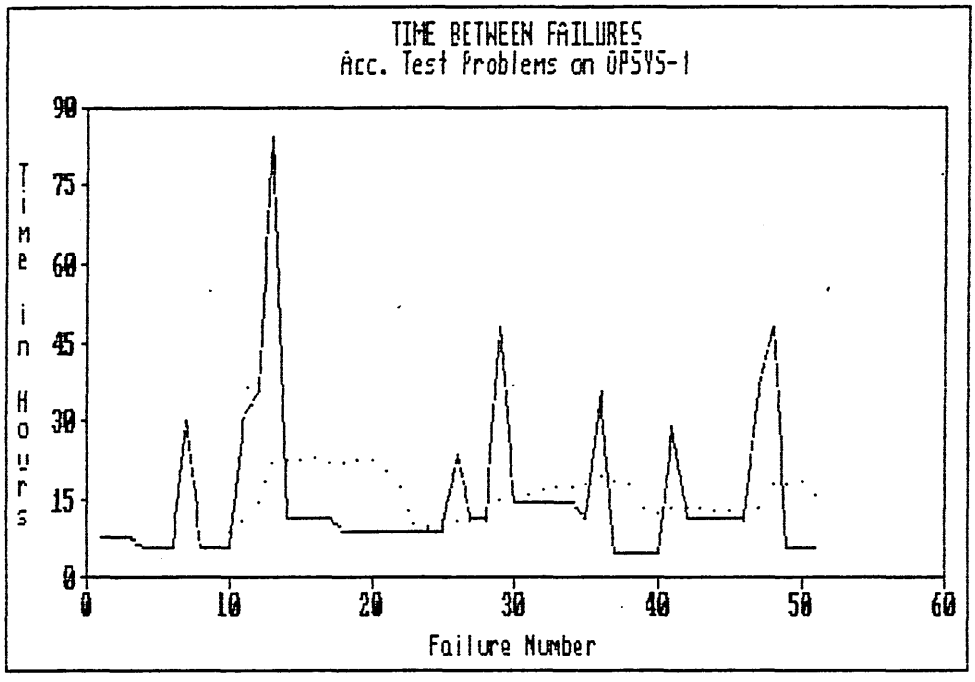
If we now look at the time between failure plot for the continuous test period as shown in Figure 5.14, from start of integration test through to end of acceptance test, there still appears to be a trend of decreasing time between failures. The additional dotted line,in the graph below, shows the mean time between failures, calculated as a running average over the previous 10 failures, and as can be seen it remains fairly steady over the final period. It should be noted however, that this mean time between failure should be taken as a rough guide only, because as previously discussed, a mean value from a skewed distribution is very much influenced by high values which have small probabilities - the time between failures is assumed to be from an exponential distribution.

Figure 5.14



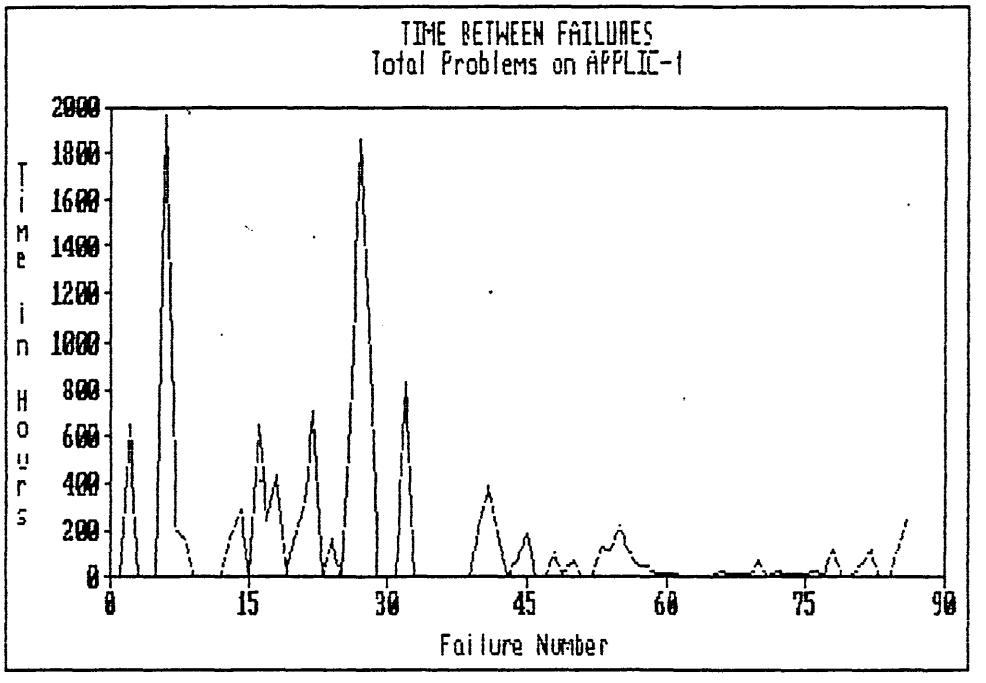
The graph below, Figure 5.15, breaks down this period further and only plots failures reported during the acceptance test phase. As can be seen, the running average of mean time between failures does not show any tendency to increase, in fact dips towards the end of the period, confirming the analysis from the cumulative plot shown in the previous section.

Figure 5.15



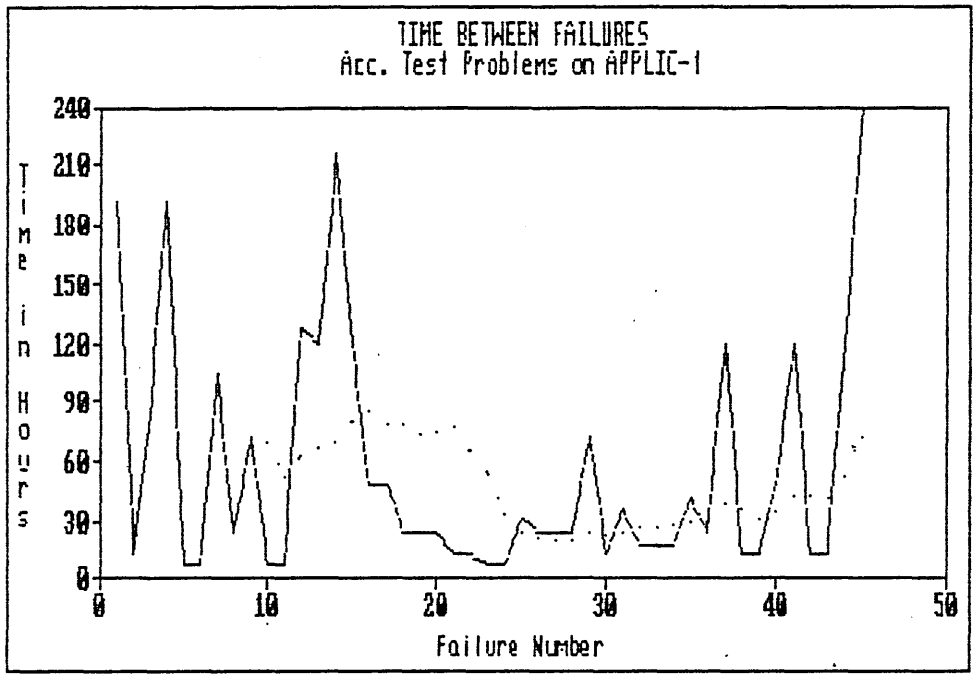
Looking next at the application product APPLIC-1, repeating the process, the following graphs show time between failures. The first graph, Figure 5.16, again plots time between failures for the total development. As with product OPSYS-1, the plot shows decreasing time between failures as the development progresses through the development cycle.

Figure 5.16



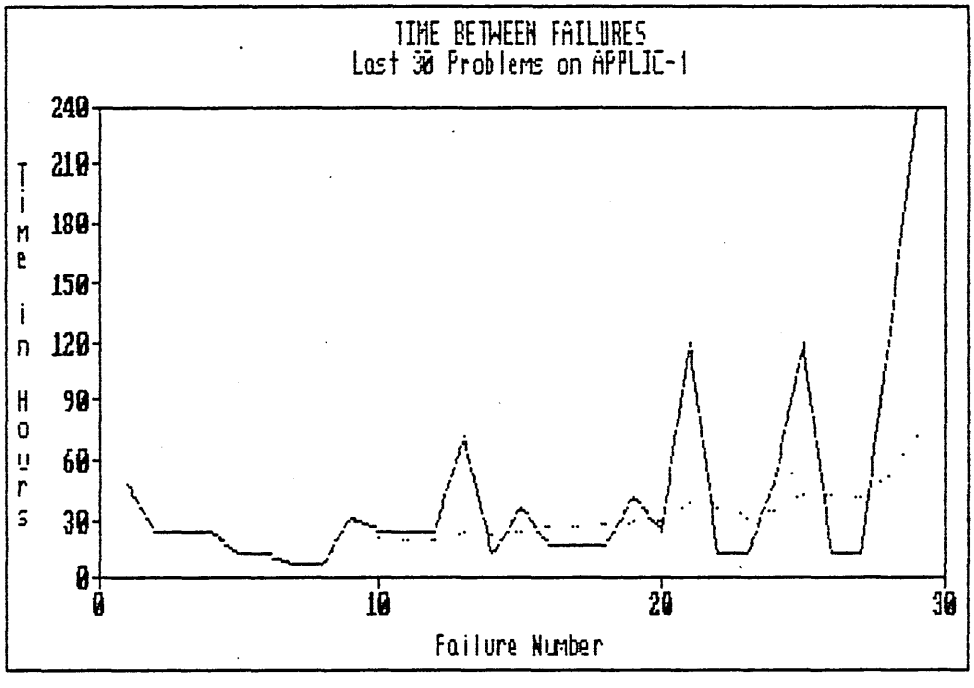
As over 50% of total failures were recorded during the acceptance test phase, this fact is not surprising. Figure 5.17 considers the time between failures, recorded during acceptance test. This plot shows the rough guide, running average of the mean time to failure, decreasing then gradually increasing towards the end of the period.

Figure 5.17



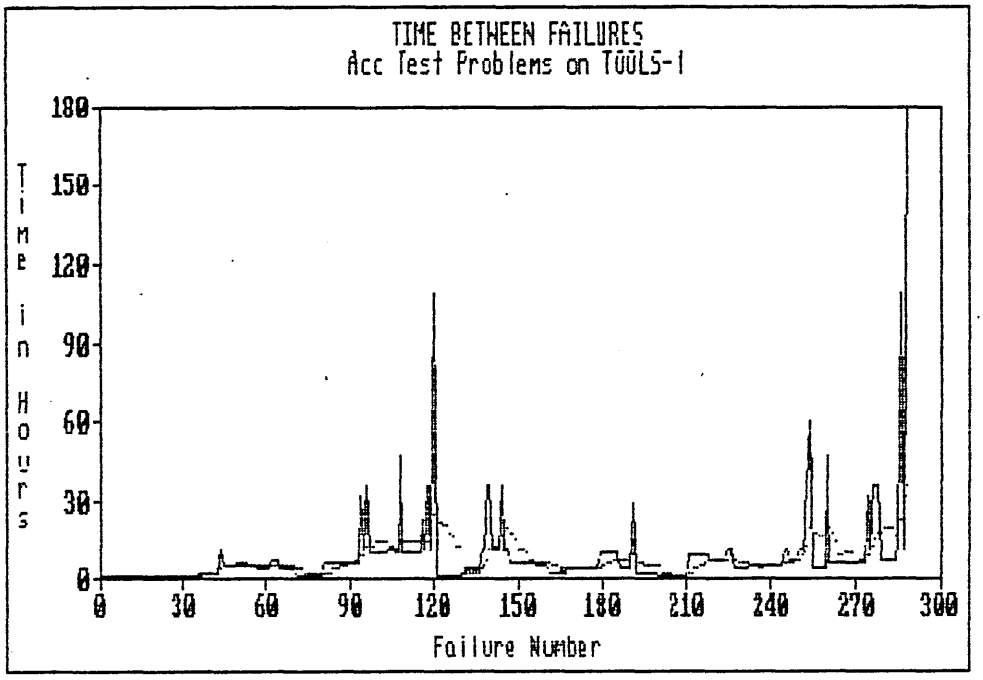
As previously stated however, this product was not consistently tested throughout the duration of the acceptance test. If only the last 30 recorded failures are analysed as shown in Figure 5.18, where the cumulative plot indicated a distinct separate failure curve, the time between failures appears to decrease then increase towards the end of the test phase. Using the running average as a rough guide of the trend, more clearly shows the gradual increase in time between failures.

Figure 5.18



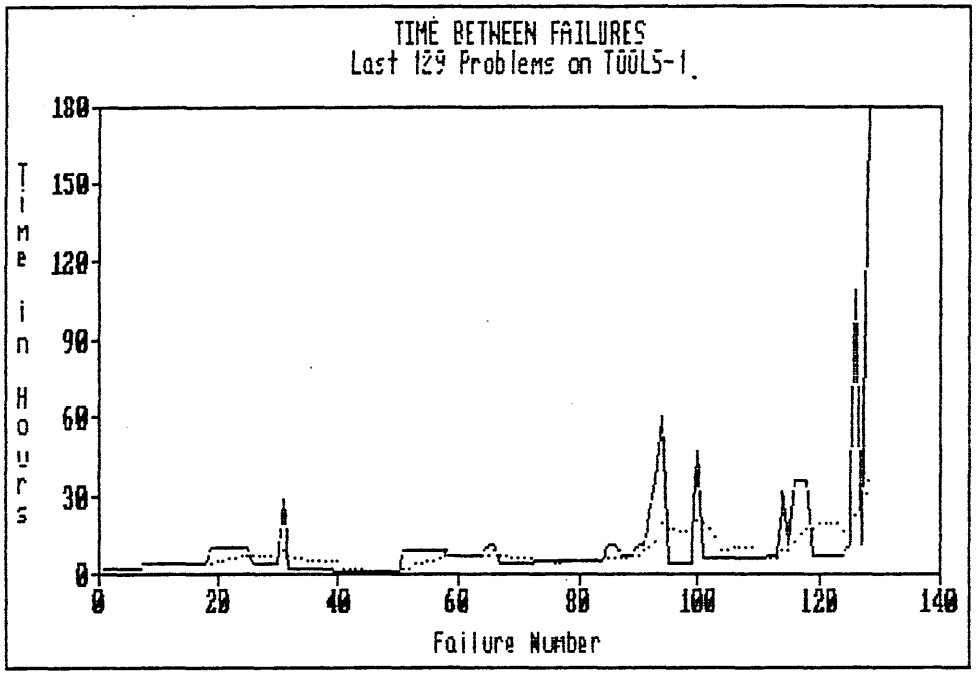
The final two graphs show time between failures for the tools product TOOLS-1. Figure 5.19 shows all failures recorded during the acceptance test phase and Figure 5.20 shows the final acceptance test phase where the cumulative curve indicated a distinct separate curve. As can be seen, neither graph is conclusive in showing a tendency towards significant increasing time between failures.

Figure 5.19



Note that this section does not include TBF's for the extension tools product. As only eleven failures were recorded during the acceptance test phase no conclusions could be drawn from the analysis.

Figure 5.20



The graphs in this section indicate that the software products do not demonstrate significantly increasing time between failures, which would show reliability growth characteristics. The running average of mean time to failure for the application and tools products show a tendency for increasing time between failures. While this increase does not appear to be significant it confirms the conclusions from the analysis of the cumulative plots, discussed in the previous section, where the application and tools products did show a tendency to level off at the end of the development phase. In the last two sections of this chapter statistical techniques will be used and the results compared with the graphical analysis and gross failure measurements.

5.2 RELIABILITY GROWTH TEST

In the previous section simple data analysis techniques were employed, comparing failure data of similar projects and producing gross failure figures and basic graphs, in an attempt to identify trends within the data. In this section, and the following section, the statistical techniques described in chapter 3, will be used to analyse product failure data.

Here we look at Raftery's [14] test for reliability growth. Considering the recent operating system OPSYS-1 release first and applying the procedure to failure reports during the integration/acceptance test phase, it was found that there was significant negative reliability growth - producing a Bayes factor of 523. This figure is not surprising considering the graphs of the time between failures presented in section 3.1.3, which shows decreasing TBF's towards the end of the test phase. If the test is applied to failures reported during the acceptance test phase, a Bayes factor value of 103 is obtained, again indicating evidence of negative reliability growth. This reinforces the view obtained from the graphical analysis, which indicated no levelling off of the cumulative failure curve nor a tendency of increasing time between failures.

If Raftery's test for reliability growth is applied to the recent application APPLIC-1 release, applying the

procedure to failure reports during acceptance test, it was again found that significant negative reliability growth was apparent - producing a Bayes factor of 104. Again, this figure is not surprising considering the graphs of the time between failures presented in section 3.1.3, which appeared to show a swing from decreasing TBF's to increasing TBF's. If the test is applied to the last 30 failure records a Bayes factor value of 0.96 is obtained, indicating evidence of reliability growth. However, as previously discussed, a rough order of magnitude interpretation suggests that reliability growth should only be regarded as strong if the Bayes factor is less than 0.1.

Applying the Raftery test to failures reported during acceptance test on the software tools product TOOLS-1, it was found that the last three failures was demonstrating decisive reliability growth, with a Bayes factor of less than 0.01. When the measure was also applied to previous failures, the Bayes factor for the previous eight records were found to demonstrate strong reliability growth, less than 0.1. While the graphs,in the previous section, of time between failures were not conclusive, this confirms the impression from the levelling off of the cumulative plot in section 3.1.2, indicating an increase in the time between failures.

5.3 LITTLEWOOD-VERRALL MODEL

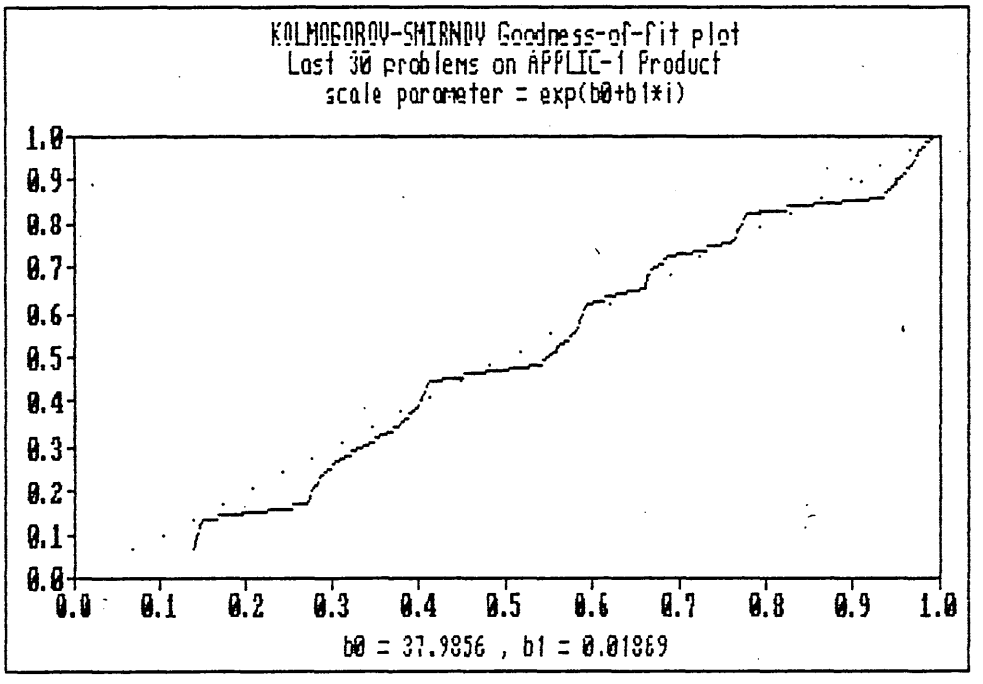
In this section the Littlewood and Verrall Bayesian reliability model, discussed in section 3.1, will be applied to failure data, recorded during the acceptance test phase.

If the recent operating system (OPSYS-1) is considered first. Comparing the results obtained from the graphical analysis in sections 5.1.2 and 5.1.3, coupled with Raftery's reliability growth test; there appears to be no evidence of reliability growth. Applying the model, using the specified scale parameter function : $\Psi(i)=\exp(\beta_0+\beta_1 i)$, it was found that the value for β_1 that minimised the Kolmogorov-Smirnov goodness-of-fit statistic was a negative value. This invalidates the model assumption that $\Psi(.)$ is a monotonically increasing function of i, which indicates reliability growth over time - if β_1 is negative $\Psi(i) > \Psi(i+1)$. No estimates for the model parameters, therefore, can be obtained and questions the validity of the release decision.

When the Littlewood-Verrall Bayesian reliability model is applied to the recently released application product (APPLIC-1), using failure data recorded during over the complete acceptance test phase, the estimated β_1 value was again found to be negative. It does however confirm the result obtained from section 5.2, the reliability growth test of Raftery, which indicated negative reliability

growth for failure reports on the complete acceptance test phase. When the L-V model is applied to the last 30 failure reports , during acceptance test, the parameter values obtained for (β_0,β_1) were (37.9856,0.01869) with a Kolmogorov-Smirnov goodness-of-fit value of 0.071. The U-plot, using these values, is shown in the following graph.

Figure 5.21



The form of scale parameter function was then changed to determine if any improvement in the model fit could be achieved, as the cumulative plot of failure data did not suggest strong reliability growth. The functions used were linear ($\beta_0+\beta_1 i$) and quadratic ($\beta_0+\beta_1 i^2$) but as can be seen from the figures below the results were significantly worse and are demonstrating bias.

Figure 5.22

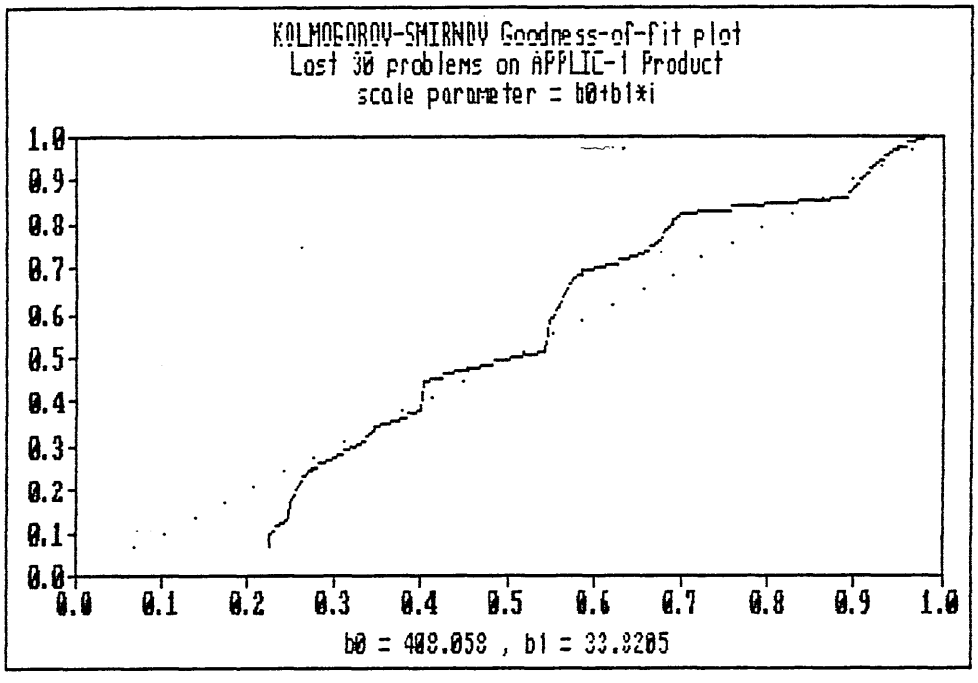
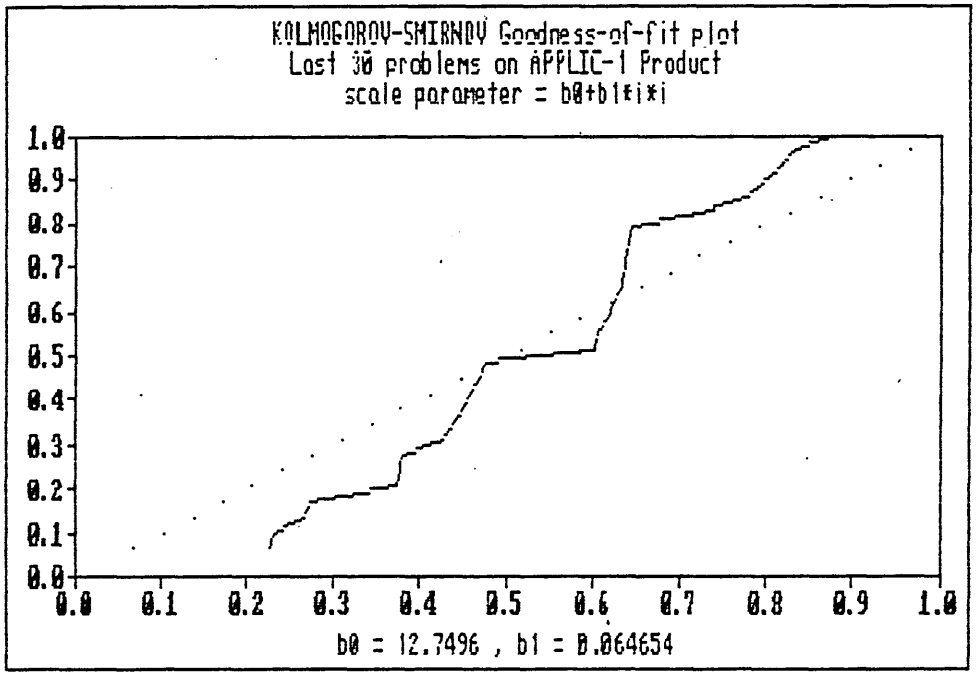
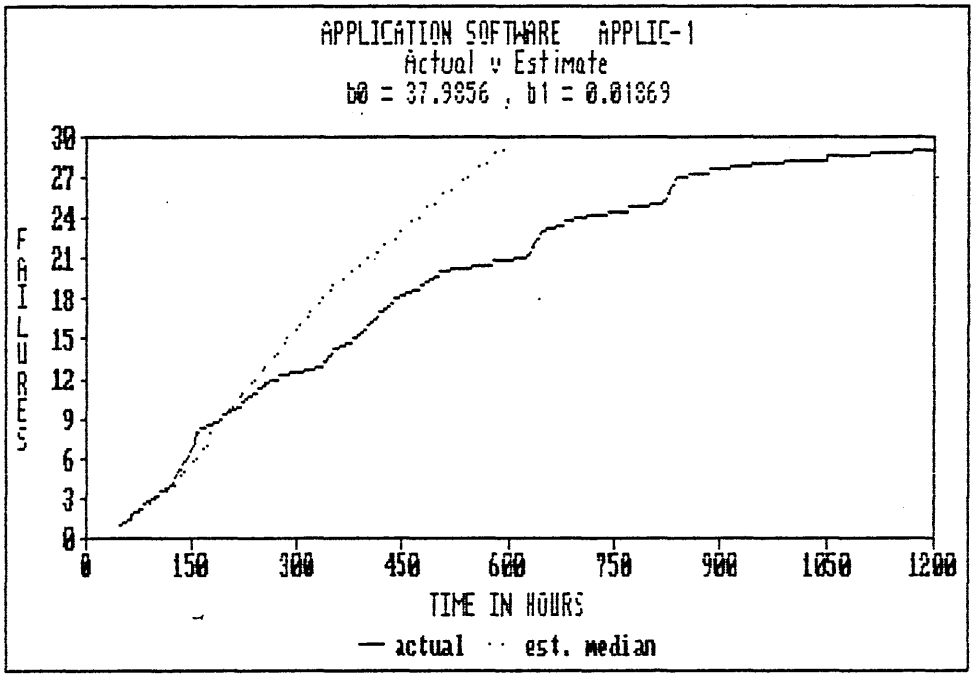


Figure 5.23



The estimates referred to in figure 5.21 can now be used to predict the median time to next failure. The estimated values can be compared against actual values to determine the accuracy of these estimates. This was achieved by evaluating the median time to next failure for each failure point, using previous actual failure data, as described in section 3.2. The plot of actual failures compared to estimated failures is shown in figure 5.24. As can be seen the estimated median failure times do not compare favourably with the actual failure data.

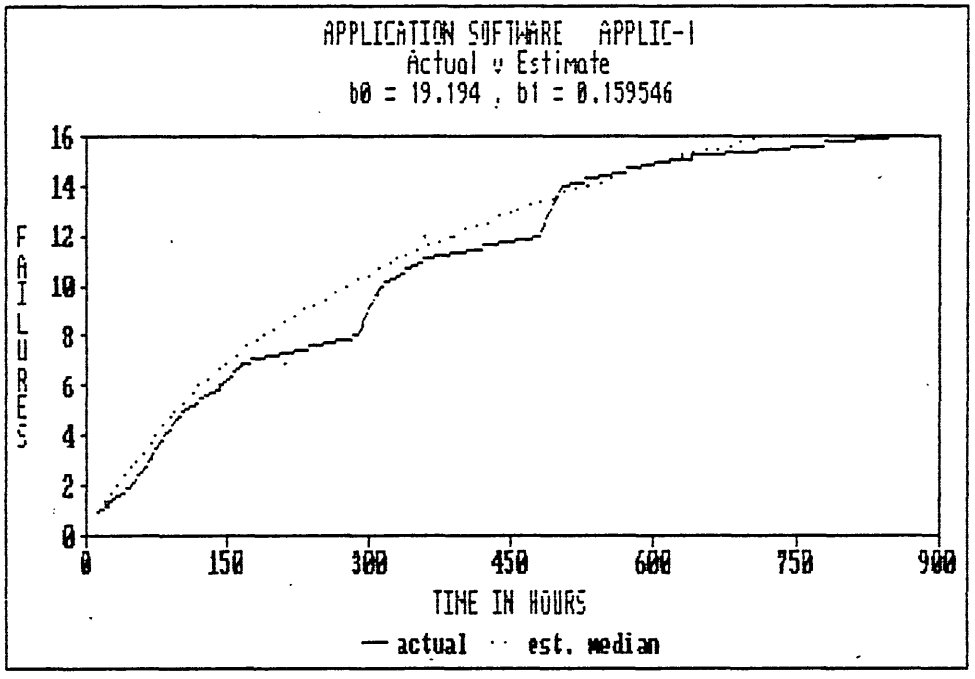
Figure 5.24



Various start and step values for the search procedure were tried and other goodness-of-fit statistics used but the parameter estimates converged back to these values or other values that produced equally poor results.

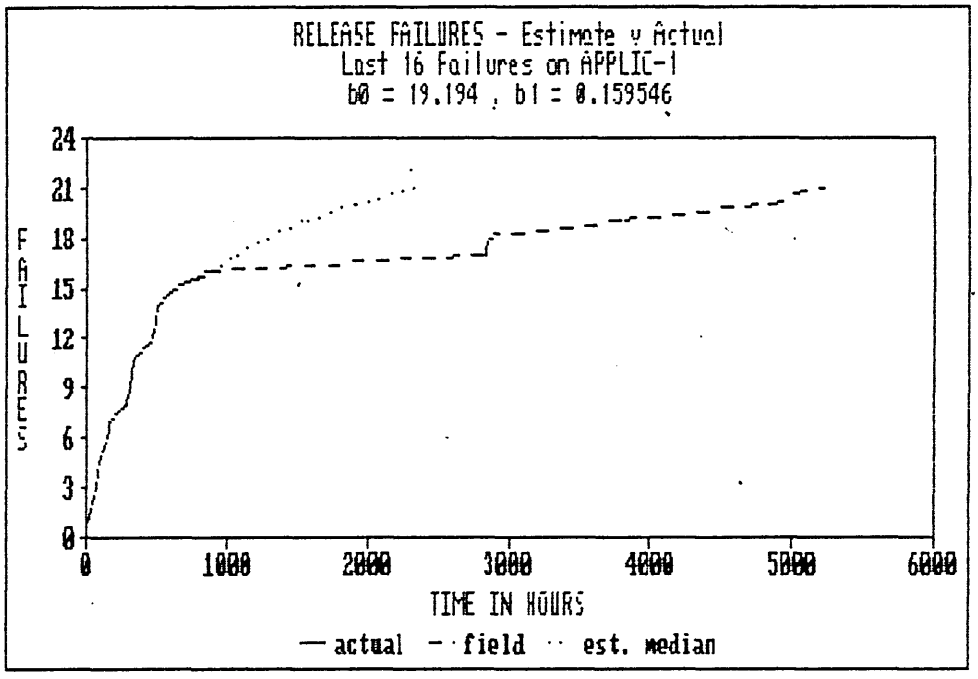
The analysis was then conducted on the last 16 failure reports as there is an indication, from the cumulative plot, of a different curve at the end of the reporting period. When the L-V model is applied to the last 16 failure reports the parameter values obtained for (β_0, β_1) were (19.194,0.159546) with a Kolmogorov-Smirnov goodness-of-fit value of 0.047. Figure 5.25 shows the actual values plotted against the estimated median values and as can be seen, appears to be a reasonable estimation.

Figure 5.25



If the estimated median time to next failure is projected into the future, predictions of field failures can then be estimated. After six months of field running with this product, five customer complaints were recieved. The field complaints are combined with prerelease test failures and are shown in figure 5.26, along with the prediction of times to next five failures.

Figure 5.26



As can be seen, the estimated median times to failure are extremely pessimistic. This however, is not totally unexpected, as the operational environment for prerelease testing will not be the same as normal field running.

Finally, if the Littlewood-Verrall Bayesian reliability model is applied to the recently released tools product (TOOLS-1), using the last 129 failure data records during

the acceptance test phase, the parameter values obtained for (β_0,β_1) were (4.023,0.024012) with a Kolmogorov-Smirnov goodness-of-fit value of 0.174. This is a poor goodness-of-fit statistic but as can be seen from figure 5.27, the estimated median values closely match actual failure data. Again, however, when this median estimate is projected into the future, then compared against actual customer complaints, the predictions are extremely pessimistic.

Figure 5.27

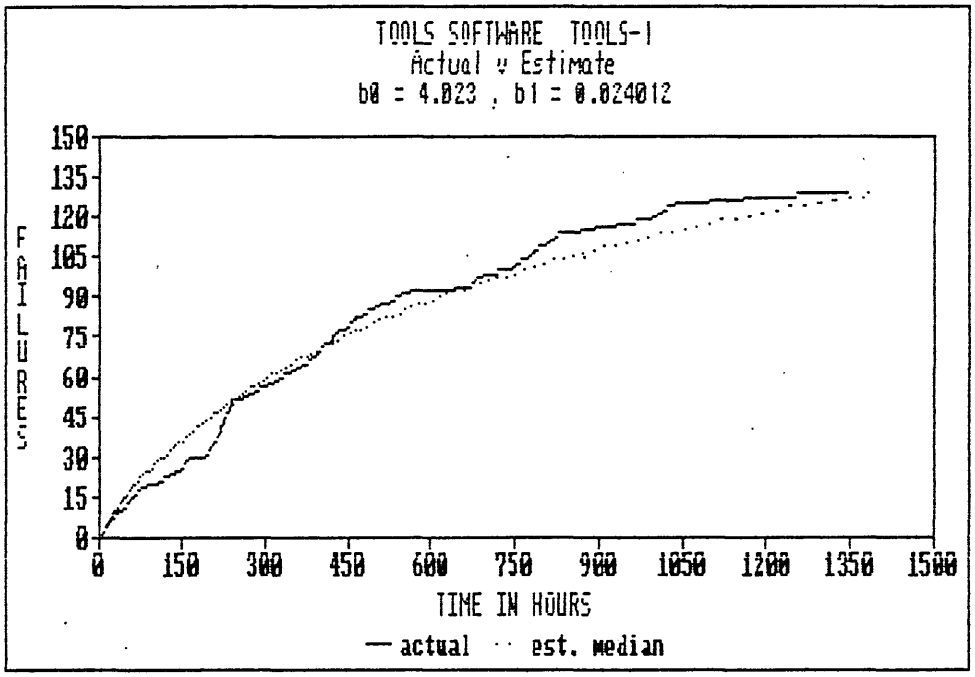
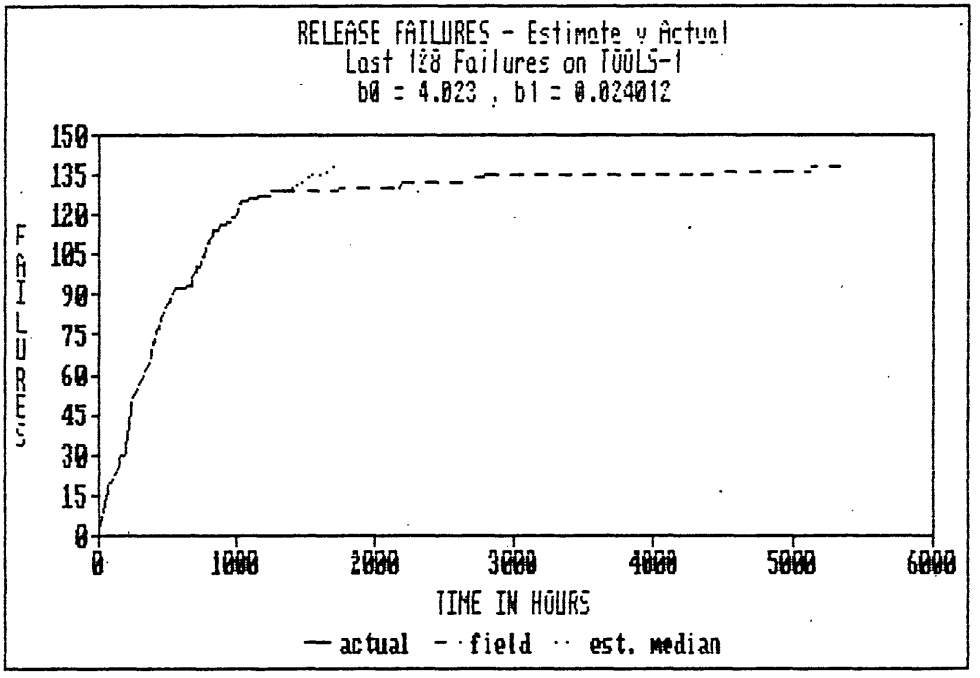


Figure 5.28 shows this plot, where nine customer failures were reported in the first six months after release.

Figure 5.28



Further product developments require to be analysed to determine if consistent relationship between pre- to post-release failure rates exist. If this can be established, a weighting factor can be included into the post-release predictions. This point is discussed further in the next chapter.

CHAPTER 6 CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

This final chapter will look at the main conclusions from this research and suggestions for future activities that should be pursued, to aid decision making throughout the development process and improve software product quality.

6.1 RELIABILITY MODELS

From the results presented in sections 3.2 and 5.3 it can be seen that the use of reliability prediction models for any decision on software quality is littered with problems and potentially misleading information. Here there has been an attempt to highlight some difficulties related to estimating the components of the scale parameter of the failure distribution; problems in selecting a goodness of fit statistic, selecting a search technique that is efficient and gives consistent results. Also, little consideration has been given here to specifying the distribution function $\psi(.)$ which is unlikely to be as defined by Littlewood and Verrall, or the simple variations tried in section 5.3 . It may not be consistent over different software projects, or consistent throughout the product development. Even if all these difficulties are resolved, it can be seen from section 3.2 that the estimates of future events are optimistic, which results in any prediction being of doubtful value.

Shooman [17] states that existing software reliability models are adequate for prediction and should now be more widely used and goes on to say that if the prediction of time between failures agrees with the field measurements within 25%, the results are excellent. While this accuracy may be good enough as a rough guide in monitoring field performance it is not within acceptable limits to base decisions on software release or planned maintenance activity, as this variability will be compounded when estimating a number of future failures. With the continual improvement in data collection and analysis it is hoped that improvement in the predictive capabilities of reliability models will be possible.

All these predictions relate to the routine running of a software product, not to the testing process, as Littlewood and Verrall state in their paper. Many other models have been suggested, including some for debugging - which have been briefly described. However, all those of which the author is aware deal with a complete software product as a unit. This approach was, perhaps, appropriate when software engineering methods were at a stage analogous to industrial quality control; the development/production process is completed and then the final product is checked for quality. But software engineers have moved on and present methods are analogous to quality assurance, every stage of the development/production process is checked and progress to the next stage is not permitted until certain criteria

have been satisfied. With increasing pressure on software engineers to achieve highly reliable products and to quantify their reliability, we need a new approach to modeling the development process and/or a new approach to decision making at each stage. This new approach must take account of and quantify the particular quality requirements of each stage of the development process and provide criteria to assist in the decision whether to proceed to the next step. Software reliability prediction models, used in conjunction with other software metrics, must be used in this new approach.

6.2 IMPROVING DATA & ANALYSIS

Software reliability models can not be used in isolation, even if the predictive ability is improved, and must be used in conjunction with other metrics that are already in use. Graphical techniques have proved useful for indicating trends in previous and current projects and should continue to be used. The introduction of Raftery's reliability growth test should also be considered, where a module or subsystem would not progress to the next test phase until there is a demonstration of strong reliability growth at the current phase. At the start of the next test phase the reliability prediction model could then be applied to failure data from the previous phase, to estimate the time to next x failures. If the failure data, at the start of this phase, is not meeting or

exceeding expectations the fact can be highlighted as early as possible allowing timely recovery action to be put in place.

Part of the reason for the poor predictive results is undoubtedly the nonrecording issue but there are also deficiencies in the failure data currently collected. There is, for example, no mandatory field for recording the development phase where the failure was detected nor a module field which would allow disaggregation of the data to analyse individual modules for patterns and exceptions. This deficiency in the database has been rectified by introducing yet another failure recording database which includes the development phase and module fields as mandatory input. This will be a local database and its use for prerelease failure recording is to be promoted. With easier access it is hoped to reduce the incidence of nonrecording. Also, if the proposal outlined in the previous section is adopted, where attention is drawn to the fact that actual failures are exceeding expected failures based on previous failure data, it will encourage failure reporting and thus ensure that records accurately reflect actual fault manifestation.

A further problem with the recorded data is the problem of multiple failures recorded on the same day which reflected the time the failure was recorded on the database and not time when the failure was detected. Future work will move away from calendar time as a parameter and more accurately

measure failures against test time, as stressed by Kruger [18]. This is similar to the point emphasized by Musa and Ackerman [19], who correctly state that if any statistics are to be used to provide quantitative guidelines software execution time needs to be accurately measured. Initial investigations into collecting accurate test effort data has begun, which has raised further questions on data interpretation. Individual members of the test team spend different amounts of time testing the software. Each team member is subjected to interruptions, attending meetings, reviews and inspections on other projects. Test effort, therefore, is variable and discontinuous. The method adopted in cumulating this test effort can be explained by the following example. As fault manifestation is assumed to be a Poisson process, start times can be ignored. Each individual's discrete test period can be combined to give total time each individual spent testing. Suppose the test team consisted of three individuals; who spent 10, 15 and 20 test hours respectively. The time between failures for each fault, up to 10 hours would be multiplied by three, to reflect the test effort that was expended prior to a failure being detected. After 10 hours, each failure up to 15 hours would be multiplied by two and after 15 hours the time between each failure would be recorded as actual. This method, therefore, accurately records time between failures against total test effort. Initial results in this area are encouraging and investigation will continue.

6.3 IMPROVING THE PROCESS

Software reliability metrics, including reliability prediction models, will show failure data trends throughout the development process and may be used to compare separate product developments. Collecting and recording these metrics, however, will not improve product quality. What will improve quality is refinements and conformance to the development process. There are three main interrelated areas that will lead to improvements, categorized as requirements, schedules and failure discovery profile. The first point is that detailed requirement specifications are required. Currently a top level product requirements specification is produced. As this document does not break down the requirements to the lowest level, the size of the development task is often not fully appreciated. This leads to schedules being defined that are impossible to achieve. As the development progresses the implementation reaches a wider audience who have a different interpretation of these nonexistent detailed requirements, leading to an increase in the level of requests for change to functionality. This unplanned activity requires documentation and code updates, adding unforeseen and unplanned effort into the development. With this increased schedule pressure the formal inspection and test processes become less efficient and at times are omitted entirely. This in turn leads to an increase in the number of faults being found late in the development cycle, with the extra costs and effort

that fault removal entails.

This is an extreme case, but highlights the need for detailed requirement documentation. It would allow a more accurate estimation of the development effort required and hence more realistic schedules. With a better understanding of requirements, unplanned change requests would also reduce, in turn reducing the urge to deviate from the development process, enabling faults to be discovered earlier in the development cycle.

In 1987 the division set out overall software measures that were basic and simple to collect, to focus developers towards discovering failures earlier in the development cycle and reduce the overall problem discovery. From the failure data reported in section 5.1.1 it can be seen that only the recent operating system came anywhere near meeting these targets. NCR Self Service Division produce products of high quality and reliability, which is borne out by the field reported failures, discussed in the previous chapter. For continued success a comprehensive metrics programme must be incorporated into the process and must be closely monitored. The success of a metrics programme, however, not only needs the cooperation of the developers, who must record potentially sensitive data, but needs support and commitment from management at the highest level.

6.4 SUGGESTIONS FOR FURTHER WORK

Future activities will continue to refine and improve the use of reliability prediction models, and investigate the suitability of new models as they are developed, within a complete metrics programme, as part of an overall process improvement strategy. The problems associated with the use of reliability prediction models, discussed in an earlier section of this chapter, exist when using the model on 'good' failure data. The problems will be magnified when the quality of the failure data is questionable.

While this research has not been completely successful in its original aim - to use software reliability prediction models as an aid to decision making for progressing the product through the development process - it has reinforced the view that metrics, not just quality related metrics, require to be collected and used throughout the software development process. The use of software metrics will allow a better understanding of the software development process, closely monitor product development and identify exceptions as soon as they arise. Once established, software metrics will allow the measurement of the effects of process improvement and effectiveness of the introduction of new development tools. Software metrics will make management decisions less subjective and form the basis for objective reasoning and decision making throughout the development of software products.

Reliability prediction models are only one tool available in the software development environment, to control and measure software projects. Future activities will investigate and introduce a wider range of software metrics, in conjunction with process improvements. This should begin with a review of the work already undertaken by Grady and Caswell [16], DeMarco [20], Kitchenham and Walker [21] to identify the metrics that best suit the company's particular needs and requirements.

To date there has not been sufficient data recorded from formal documentation and code inspections on completed projects to observe the expected benefits of a reduction in the number of failures found during test phases. This activity is also expected to improve the predictive capabilities of prediction models, which is a view shared by Chang et al. [22] who state that design reviews or code walkthroughs are necessary before actual reliability testing is performed.

A recently acquired CASE tool that introduces formal structured design and analysis techniques into the planning and design phases of the development will overcome the missing requirement problem. Pilot projects have been identified to evaluate the methodology. As this breaks down the analysis and design activities into well defined areas, in parallel, productivity metrics will be defined, measures taken and analysed. These measures,

coupled with complexity metrics [23] that will help to define test strategies and allow test effort to be more closely estimated, will ensure that development schedules for future projects will be more accurately estimated, allowing realistic schedules to be established.

Problems in projecting the predictions from pre- to post-release have not yet been successfully overcome. Pre-release testing does not reflect the typical operational environment. The test philosophy adopted at NCR's Self Service Systems Division is to stress test all software products, from module through to system test, systematically validating each functional requirement. Future work, once sufficient projects relating failures to test effort have been collected, will look at applying a weighting factor to the predictions that will take into account this atypical input domain.

This thesis concludes with the statement that the job is just beginning, the hard work lies ahead. For the company to remain the best in its field it must continue to produce successful, reliable products. The only way that this will continue to be achieved, as the products become more sophisticated and complex, is to understand and improve the development process. The only way to fully understand the process is to measure it.

APPENDIX A - LITTLEWOOD AND VERRALL MODEL

The following expands on the method based on Probability Integral Transforms, in the paper by Littlewood and Verrall [4], to obtain a sample distribution function from a uniform distribution.

Littlewood and Verrall show that the distribution function is given by ;

$$F(t_{n+1}) = P(T_{n+1} < t_{n+1})$$

$$= 1 - \left[\gamma \,/\, \left\langle \gamma + \ln \left[\frac{t_{n+1} + \psi(n+1)}{\psi(n+1)} \right] \right\rangle \right]^{n+1}$$

where $\gamma = \ln \prod_{i=1}^{n} (1/k_i)$, and $k_i = \{\psi(i)\}/\{\psi(i)+t_i\}$

(L-V) then use $y_{\alpha}^{(n+1)}$ to denote an upper 100α% confidence bound of this distribution,

ie $P[T_{n+1} < y_{\alpha}^{(n+1)}] = \alpha$

therefore

$$\alpha = 1 - \left[\gamma \,/\, \left\langle \gamma + \ln \left[\frac{y_{\alpha}^{n+1} + \psi(n+1)}{\psi(n+1)} \right] \right\rangle \right]^{n+1}$$

$$\left[1-\alpha\right]^{1/(n+1)} = \frac{\gamma}{\gamma + \ln \left[\frac{y_{\alpha}^{n+1} + \psi(n+1)}{\psi(n+1)} \right]}$$

$$\ln \left[\frac{y_{\alpha}^{n+1} + \psi(n+1)}{\psi(n+1)} \right] = \gamma \left\langle 1-\alpha \right\rangle^{-1/(n+1)} - \gamma$$

$$y_{\alpha}^{n+1} = \left[\exp\{ \gamma \left\langle 1-\alpha \right\rangle^{-1/(n+1)} - \gamma \} \right] \psi(n+1) - \psi(n+1)$$

$$= \psi(n+1) \left[\exp\{ \gamma \left[\left\langle 1-\alpha \right\rangle^{-1/(n+1)} - 1 \right] \} \right] - \psi(n+1)$$

$$Y_{\alpha}^{n+1} = \psi(n+1)\left[\exp\gamma\right]^{\left((1-\alpha)^{-1/(n+1)}-1\right)} - \psi(n+1)$$

$$= \psi(n+1)\left[\prod_{i=1}^{n}(1/k_i)\right]^{\left((1-\alpha)^{-1/(n+1)}-1\right)} - \psi(n+1)$$

$$= \psi(n+1)\left[\prod_{i=1}^{n}k_i\right]^{1-(1-\alpha)^{-1/(n+1)}} - \psi(n+1)$$

$$= \psi(n+1)\left[\prod_{i=1}^{n}k_i\right]^{1-1/(1-\alpha)^{1/(n+1)}} - \psi(n+1)$$

Looking back at equation (3), in section 3.1, we have

$$Y(x) = \frac{1}{n}\sum_{i=1}^{n} Y_i(x)$$

$$\text{therefore}\quad E[Y(x)] = E\left[\frac{1}{n}\sum_{i=1}^{n} Y_i(x)\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n} E[Y_i(x)]$$

$$= \frac{1}{n}\sum_{i=1}^{n} P[Y_i(x)=1]$$

$$= \frac{1}{n}\sum_{i=1}^{n} P[T_i < t_i(x)] = x$$

t_i is the time from the (i-1)th to (i)th failure,

and x_i is the solution of $t_i = t_i(x)$

where $P[T_i < t_i(x)] = x$

So if ψ is completely determined, we have an equation for x_i given by;

$$t_i = \psi(i)\left[\left(\prod_{m=1}^{i-1}k_m\right)^{1-1/(1-x_i)^{1/i}} - 1\right]$$

$$\frac{t_i}{\psi(i)} + 1 = \left[\prod_{m=1}^{i-1}k_m\right]^{1-1/(1-x_i)^{1/i}}$$

note that $\dfrac{t_i}{\psi(i)} + 1 = \dfrac{1}{k_i}$

therefore $-\ln k_i = \left[1 - 1/(1-x_i)^{1/i} \right] \sum_{m=1}^{i-1} \ln k_m$

$$1 + \frac{\ln k_i}{\sum_{m=1}^{i-1} \ln k_m} = \frac{1}{(1-x_i)^{1/i}}$$

$$\frac{\sum_{m=1}^{i} \ln k_m}{\sum_{m=1}^{i-1} \ln k_m} = \frac{1}{(1-x_i)^{1/i}}$$

therefore, $x_i = 1 - \left[\dfrac{\sum_{m=1}^{i-1} \ln k_m}{\sum_{m=1}^{i} \ln k_m} \right]^i$

APPENDIX B - RAFTERY'S RELIABILITY GROWTH TEST

This appendix is a detailed explanation of Raftery's debugging model introduced in section 3.3 .

The model assumptions are ;

System observed for period [0,T], during which n failures have occurred at times $t=(t_1, \ldots, t_n)$, where $n>1$.

Sample space consists of systems, therefore N (total number of faults) is a random variable.

N has a Poisson distribution, equivalent to a non-homogeneous Poisson process with rate function

$$M_1: \lambda(s) = \rho \exp(-\beta s)$$

where $\lambda(s)$ is the rate of occurrence of failures at time s, ρ and β are unknown parameters ($\rho>0$ & $E[N]=\rho/\beta$)

Testing for Reliability Growth

The paper compares the distribution M_1 with a constant rate Poisson process

$$M_0: \lambda(s) = \mu$$

The comparison of M_0 with M_1 is based on the Bayes Factor,

$$B_{01} = p(t|M_0)/p(t|M_1)$$

the ratio of marginal likelihoods.

Where, $$p(t|M_0) = \int_0^{\infty} p(t|\mu,M_0)\,p(\mu|M_0)\,d\mu$$

$$p(t|M_1) = \int_0^{\infty}\int_0^{\infty} p(t|\rho,\beta,M_1)\,p(\rho,\beta|M_1)\,d\rho d\beta$$

The paper gives (the standard vague prior for μ)

$$p(\mu|M_0) = C_0\mu^{-1} \qquad \text{(Jaynes [24])}$$

$$p(t|\mu,M_0) = \frac{\exp(-t_1/\mu)}{\mu} \times \frac{\exp(-(t_2-t_1)/\mu)}{\mu} \times \ldots \times \frac{\exp(-(T-t_{n-1})/\mu)}{\mu}$$

therefore,

$$p(t|M_0) = \int_0^{\infty} \frac{\exp(-T/\mu)}{\mu^n}\,\frac{C_0}{\mu}\,d\mu$$

$$= C_0 \int_0^{\infty} \frac{\exp(T/\mu)}{\mu^{(n-1)}}\,d\mu$$

Substituting $\mu = 1/z$ & $d\mu = -1/z^2\,dz$

gives $$p(t|M_0) = C_0 \int_0^{\infty} z^{(n-1)} \exp(-Tz)\,dz$$

Substituting $Tz = x$ & $Tdz = dx$

gives $$p(t|M_0) = C_0 \int_0^{\infty} (x/T)^{(n-1)} \exp(-x)\,dx/T$$

$$= C_0/T^n \int_0^{\infty} x^{(n-1)} \exp(-x)\,dx$$

$$= C_0/T^n\,\Gamma(n)$$

$$= C_0/T^n\,(n-1)! \qquad \text{B.1}$$

The paper gives

$$p(\rho,\beta|M_1) = C_1\rho^{-2} \qquad \text{(Akman \& Raftery [25])}$$

and the likelihood for M_1 as

$$p(t|\rho,\beta,M_1)=\rho^{n}\exp(-\beta S-\rho\beta^{-1}(1-\exp(-\beta T)))$$ (Bartholomew [26])

where $S=\sum_{i=1}^{n} t_i$

therefore,

$$p(t|M_1) = \int_0^{\infty}\int_0^{\infty} C_1\rho^{(n-2)}\exp(-\beta S-\rho/\beta+\rho\exp(-\beta T)/\beta\ d\rho\ d\beta$$

$$= \int_0^{\infty} C_1\exp(-\rho S)\left[\int_0^{\infty}\rho^{(n-2)}\exp(-\rho(1/\beta-\exp(-\beta T)/\beta))\,d\rho\right]d\beta$$

Substituting $-\rho(\exp(-\beta T)/\beta-1/\beta) = U$

& $-d\rho(\exp(-\beta T)/\beta-1/\beta) = dU$, gives

$$p(t|M_1) =$$

$$\int_0^{\infty} C_1\exp(-\beta S)\left[\int_0^{\infty}\left(\frac{1}{\beta}-\frac{\exp(-\beta T)}{\beta}\right)^{-(n-2)} U^{(n-2)}\exp(-U)\left(\frac{1}{\beta}-\frac{\exp(-\beta T)}{\beta}\right)^{-1} dU\right]d\beta$$

$$=\int_0^{\infty} C_1\exp(-\beta S)\left[\frac{1}{\beta}\left(1-\exp(-\beta T)\right)\right]^{-(n-1)}\Gamma(n-1)\ d\beta$$

Substituting $\beta T = y$ & $T d\beta = dy$, gives

$$p(t|M_1) = C_1(n-2)!\int_0^{\infty}\frac{\exp(-Sy/T)\,(y/T)^{(n-1)}}{(1-\exp(-y))^{(n-1)}}\frac{dy}{T}$$

$$= C_1(n-2)!/T^{n}\int_0^{\infty}\frac{y(n-1)\exp(-Ry)}{(1-\exp(-y))^{(n-1)}}\,dy \qquad \text{B.2}$$

note, $R = S/T$

Therefore, from B.1 & B.2

$$B_{o1} = \frac{C_o/T^{n}\ (n-1)!}{C_1(n-2)!/T^{n}\int_0^{\infty} y^{(n-1)}\exp(-Ry)/(1-\exp(-y))^{(n-1)}dy}$$

$$= (C_o/C_1)(n-1)\left[\int^{\infty}\exp(-Ry)\left(y/(1-\exp(-y)\right)^{(n-1)}dy\ \right]^{-1}$$

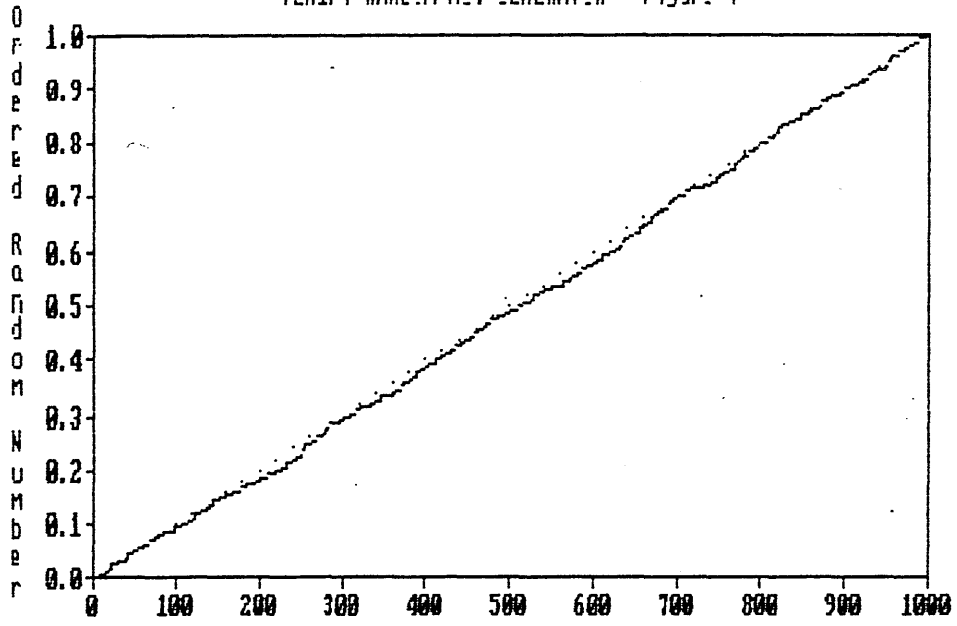
APPENDIX C - VALIDATION OF THE RANDOM NUMBER GENERATOR

Sampling distributions of the estimated parameters were obtained using random variate generation. Outlined below are the tests performed to verify the validity of the random number generator used to demonstrate that the random number stream is uniformly and independently distributed.

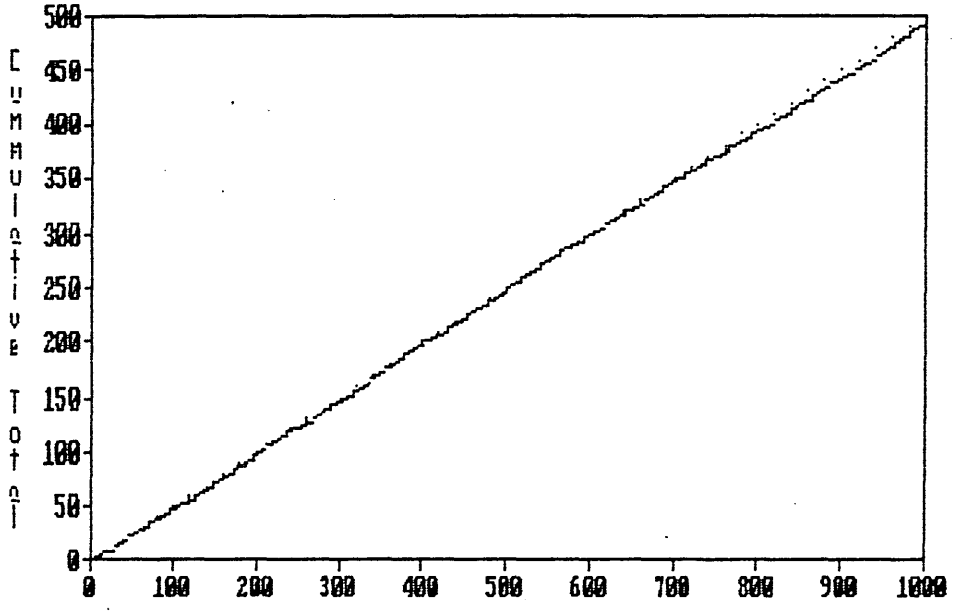
The first stage was to generate 1000 random numbers in the range 0 to 1. These were then sorted into ascending order and plotted against a line of unit slope, Figure 1. The figure shows little variation from the line of unit slope, demonstrating that the random number stream was uniformly distributed. Figure 2 shows a cumulative plot of the raw data values and as can be seen does not deviate significantly from the expected values.

To demonstrate the absence of autocorrelation, the values are independently distributed, random value I was plotted against random value I+N. Figures R3a through R3d shows this, with N = 1 through 4 respectively. The plots show points scattered randomly throughout the available space, suggesting that the values were independently distributed.

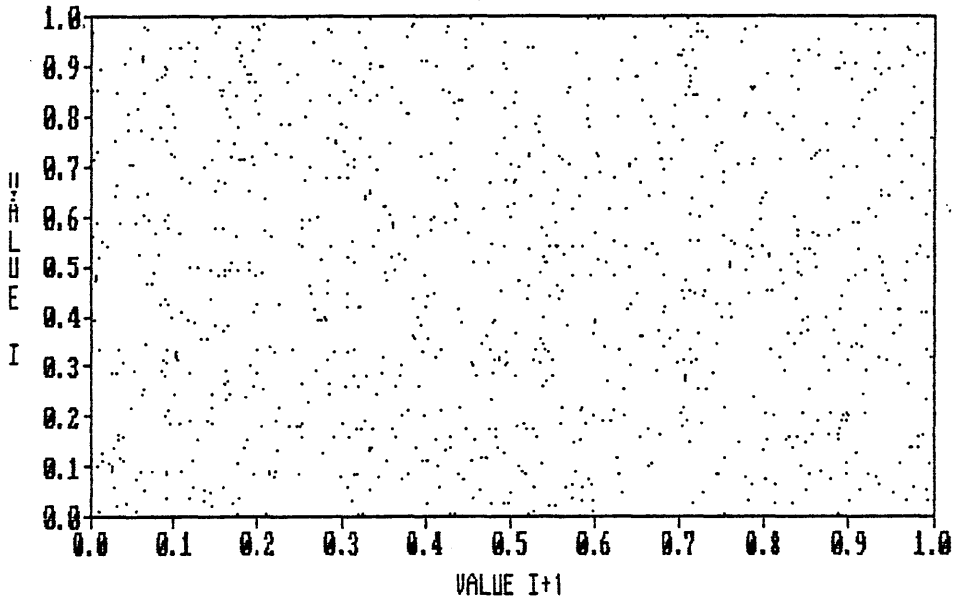
VERIFY RANDOM No. GENERATOR - Figure 1



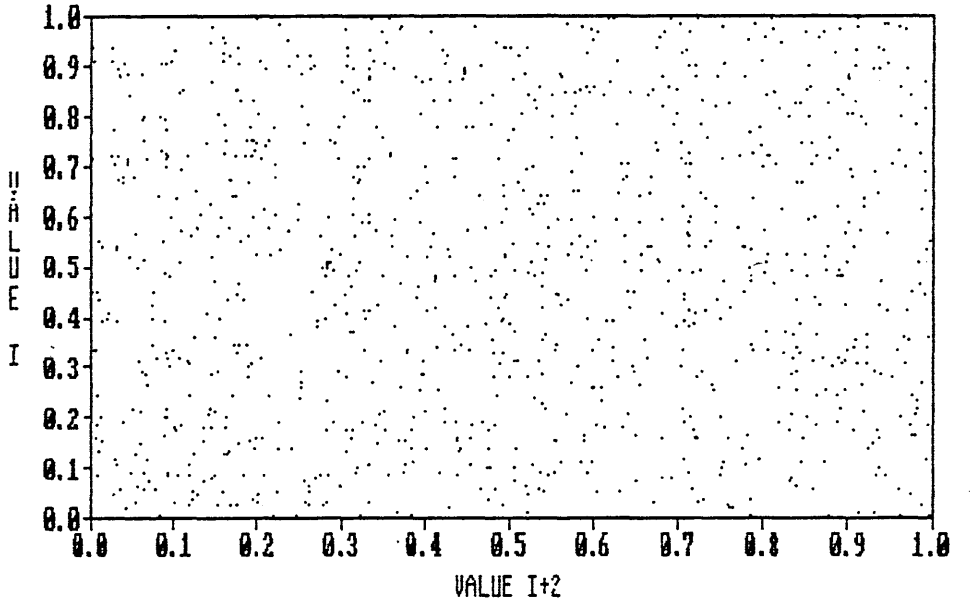
VERIFY RANDOM No. GENERATOR - Figure 2



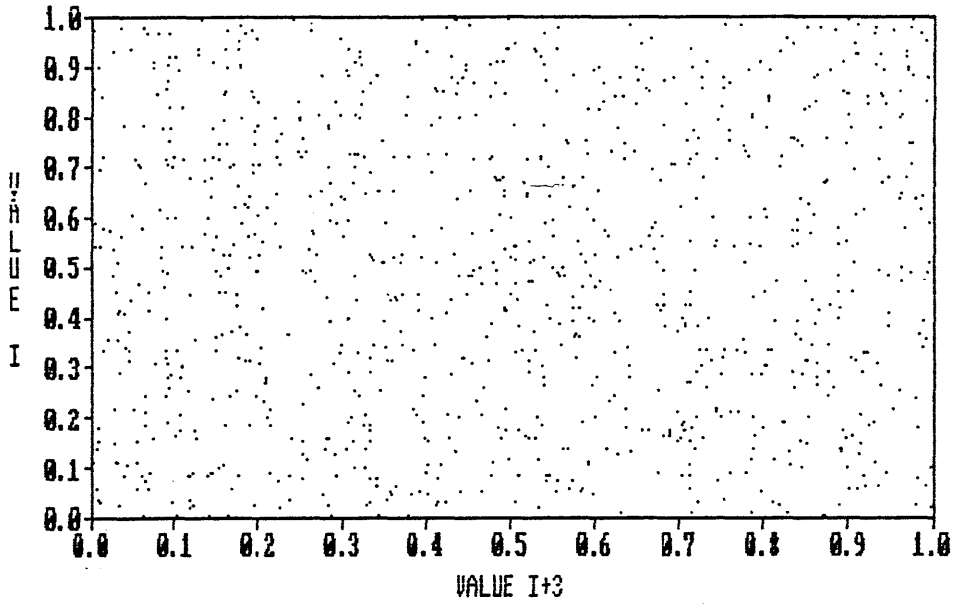
AUTOCORRELATION TEST - Figure R3a



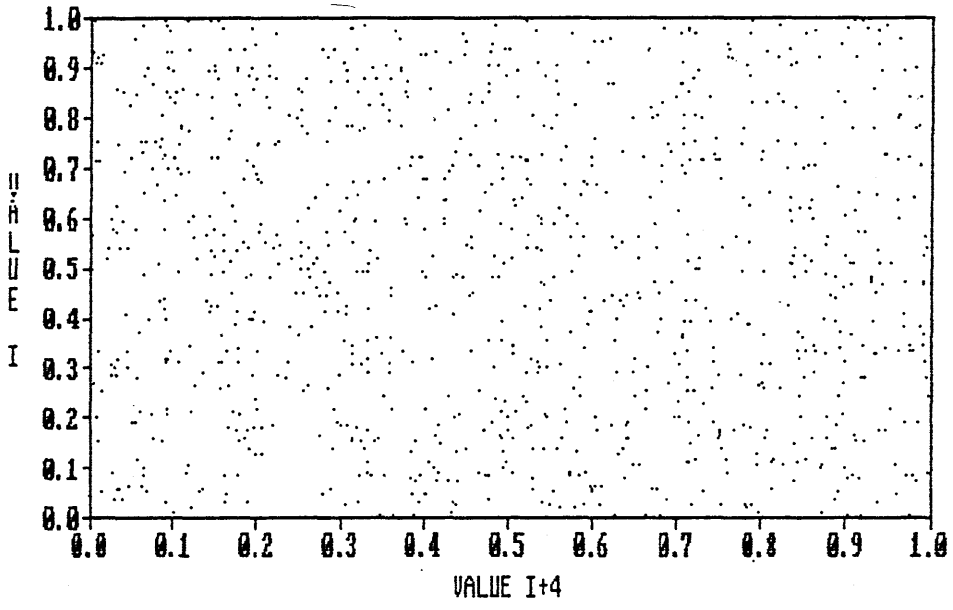
AUTOCORRELATION TEST - Figure R3b



AUTOCORRELATION TEST - Figure R3c



AUTOCORRELATION TEST - Figure R3d



The final set of validation tests, use the chi-square goodness-of-fit test technique to verify the uniformity of successive sets of numbers in the sequence [27]. The 1000 random numbers are partitioned into 10 groups of 100. A frequency table was then set up, for each group to record the occurrence within 0.1 intervals. The table, showing grouped data in columns, is given below.

	gr1	gr2	gr3	gr4	gr5	gr6	gr7	gr8	gr9	gr10
0.000 to 0.099	12	6	11	5	14	12	7	15	13	13
0.100 to 0.199	11	18	9	15	10	11	11	10	5	15
0.200 to 0.299	7	11	10	8	7	8	7	10	9	6
0.300 to 0.399	12	8	12	10	10	11	12	7	13	11
0.400 to 0.499	8	9	11	10	10	10	13	11	16	4
0.500 to 0.599	16	5	10	8	11	7	13	17	15	7
0.600 to 0.699	7	8	9	7	4	9	11	8	4	9
0.700 to 0.799	13	13	11	15	9	8	12	8	5	9
0.800 to 0.899	8	15	9	10	9	12	10	5	12	12
0.900 to 0.999	6	7	8	12	16	12	4	9	8	14

The first test compared each group with the expected value, a uniform distribution with 10 occurrences in each interval. The tabulated value [28], for $X^2_{0.05,9}$, is given as 16.919. The chi-square statistic for each group is given below.

gr1	gr2	gr3	gr4	gr5	gr6	gr7	gr8	gr9	gr10
9.60	15.80	1.40	9.60	10.00	3.20	8.20	11.80	17.40	11.80

It can be seen that only one value, group 9, is greater than the tabulated value.

If each group is then taken with the sum of the groups on the right, to give a value of 9 for the degrees of freedom, the tabulated value for $X^2_{0.05,9}$ is 16.919 as above, and the calculated chi-square statistics are given in the table below.

gr1	gr2	gr3	gr4	gr5	gr6	gr7	gr8	gr9
6.65	15.45	2.33	10.94	6.11	3.98	10.57	8.16	20.58

This partitions the 81 degrees of freedom for the test of association in the full 10x10 table. As can be seen, only one statistic, from the above calculations is in the critical region. This picks up group 9 again, as in the previous test.

The above tests have all produced satisfactory results, except that group 9 of the 10 groups of 100 shows a significant departure from uniformity. It is felt however, that the use of this random number generator is justified.

APPENDIX D - SMART PROJECT FILES

```
'  ***************************************************************
'  ***************************************************************
'  ***************************************************************
' Project file  -  hj.pf1
'
'  This project file requests a text input file, containing time
'  between failures as real values  and using the Littlewood Verrall
'  model outputs estimated parameter values for a sample distribution
'  function given, initial estimates for B0 & B1 their initial step
'  values.
'  ***************************************************************

'  ***************************************************************
'  Request inputs
'    Filename - containing time between failure data
'    Initial estimates for the failure rate scale parameter
'    (Gamma dist.)
'  ***************************************************************

quiet on
repaint off
menu clear 0 11
menu print 2 20 0 11 LITTLEWOOD VERRALL Model
menu print 4 22 0 11 Enter input filename :-
menu print 5 20 0 11 Enter initial B0 value :-
menu print 6 34 0 11 B1 value :-
menu draw box 3 46 7 57 1 3
menu input 4 47 12 0 9 text1
menu input 5 47 12 0 19 $currentb0
menu input 6 47 12 0 19 $currentb1


'  ***************************************************************
'  Set up spreadsheet cell format to real numbers with 8 dec places
'  Open text file
'  Set record count variable to 0
'  ***************************************************************
value-format normal left numeric nocommas precision 8

'  ***************************************************************
'  If filename is blank then file already loaded into spreadsheet,
'  therefore find number of records and jump to start of processing.
'  ***************************************************************
if text1 = ""
  goto lower-edge
  value1 = rows(r2:c)
  $integer = 0
  jump start
endif
```

```
fopen text1 as 1
$integer = 0

'  ***************************************************************
'  Reads each record from text file, until error condition
'  (end_of_file)
'    Put  Record count ($integer)          into column 1
'         Record (time between failures)   into column 2
'    Set  value1 = number of records in file,
'         sets Record count back to 0
'         & close text file
'  ***************************************************************

@r1c1 = "FAILURE NUMBER"
r1c2  = "TIME BETWEEN FAILURES"
$integer = $integer + 1
fread 1 into $tbf

while cerror <> 3
  cursor down
  rc1 = $integer
  rc2 = val($tbf)
  $integer = $integer + 1
  fread 1 into $tbf
endwhile

value1 = $integer - 1
$integer = 0

fclose 1

'  ***************************************************************
'  Jump to start of main procedure
'       Hooke/Jeeves search algorithm
'  ***************************************************************
jump start



'  ***************************************************************
'  This procedure calls to further procedures
'       sample_distribution
'          constructs a U-plot
'       fit_xxxx
'          evaluates a goodness_of_fit value
'  Note - this file requires to be edited to change fit statistic
'  ***************************************************************
procedure step_function
  call sample_distribution
  call fit_ks
'  call fitw2
'  call fitnw2
return
```

```
'  ***************************************************************
'  input values -  $b0   :  output values -  Y(column/total) into c3
'                  $b1                       Function (X)    into c4
'                  value1
'
'  Calculates the integral transform function
'  and places values into ascending order (U-plot)
'  ***************************************************************
procedure sample_distribution

  @r2c2
  $psi = exp($b0+$b1*rc1)
  $lkm1 = ln($psi/($psi+rc))
 $sumx=0
 $rsumx=0
  while rc1 < value1
    cursor down
    $psi  = exp($b0+$b1*rc1)
    $lkm2 = $lkm1+ln($psi/($psi+rc))
    rc3 = rc1/value1
    rc4 = 1-power($lkm1/$lkm2,rc1)
    $lkm1 = $lkm2
  rc5 = ln(1-rc4)
  $sumx = $sumx + rc5
  endwhile
 @r2c2
 while rc1 < value1 - 1
  cursor down
  $rsumx = $rsumx + rc5
  rc5 = $rsumx/$sumx
 endwhile
 cursor down
blank block rc5
  sort r2:c4 ascending using column 4

return

'  ***************************************************************
'  The following finds the W^2 statistic
'  for the defined $b0 and $b1 values.
'  ***************************************************************
procedure fitw2
 $fit = 0
 $count = 1
 @r2c1

 while $count < value1
   $fit = $fit + power(rc3-rc4,2)
   cursor down
   $count = $count + 1
 endwhile
return
```

```
'  ***************************************************************
'  The following finds the nW^2 statistic
'  for the defined $b0 and $b1 values.
'  ***************************************************************
procedure fitnw2

 $fit = 0
 $count = 1
 @r2c3
 while $count < value1 - 1
   $fit = $fit + power(rc4 - (2*$count-1)/(2*value1),2)
   cursor down
   $count = $count + 1
 endwhile
 $fit = $fit + 1/(12*value1) + power(rc4 - (2*$count-1)/(2*value1),2)

return

'  ***************************************************************
'  The following finds the Kolmogorov-Smirnov statistic
'  for the defined $b0 and $b1 values.
'  ***************************************************************
procedure fit_ks

 $fit = 0
 $count = 1
 @r2c3
 while $count < value1
   $fittemp = abs(rc3 - rc4)
   if $fittemp > $fit then $fit = $fittemp
   cursor down
   $count = $count + 1
 endwhile

return


'  ***************************************************************
'  Start search for b0 & b1 values
'  that minimise goodness-of-fit statistic
'  ***************************************************************
label start

'  *******************************************
'  Set b0 & b1 to values input by User
'  *******************************************
$b1 = $currentb1
$b0 = $currentb0

'  *******************************************
'  This procedure calls sample_distribution
'   & current goodness-of-fit statistic
'  returning current fit value
'  *******************************************
call step_function
```

```
'  ********************************************
'  Set parameter step size
'  based on selected parameter size & current fit
'  (small goodness-of-fit value, small step size)
'  ********************************************
$b0_step = $b0*$fit
$b1_step = $b1*$fit


'  ********************************************
'  Start of Hooke-Jeeves search algorithm
'   set advance equal to base
'  ********************************************
r1c6 = $fit
r2c6 = $b0
r3c6 = $b1
r1c7 = r1c6
r2c7 = r2c6
r3c7 = r3c6
%1 = 1
$iteration = 0

label iteration1
$iteration = $iteration + 1
'  ********************************************
'  Draw current U-plot
'  ********************************************
 graphics generate real black/white screen fittmp

%4 = r2c6
%5 = r3c6
%6 = r1c6
%7 = $b0_step
%8 = $b1_step
%9 = $iteration

'  ********************************************
'  Display current values on screen
'  ********************************************
menu print 9 25 0 11 Start of iteration-
menu print 10 33 0 11 B1 value :-
menu print 11 36 0 11 Fit :-
menu print 12 32 0 11 B0 Step   :-
menu print 13 32 0 11 B1 Step   :-
menu print 14 32 0 11 Iteration :-
menu print 9 46 12 11 %4
menu print 10 46 12 11 %5
menu print 11 46 12 11 %6
menu print 12 46 12 11 %7
menu print 13 46 12 11 %8
menu print 14 46 12 11 %9
quiet on
repaint off
```

```
'  ********************************************
'  Search either side of current values
'    set current values from advance values
'  ********************************************
while %1 < 3
 $b1 = r3c7

'  ********************************************
'  If b1 parameter is less than zero
'     invalidates the reliability growth assumption
'     therefore set b1 just above zero
'  ********************************************
 if $b1 < 0 then $b1 = 0.0001
 $b0 = r2c7

'  ********************************************
'  if b0 parameter is greater than 34
'     then exp(b0+b1*i) tends to infinity
'     and sample distribution calculation will fail
'     therefore set b0 to 34
'  ********************************************
 if $b0 > 34 then $b0 = 34

'  ********************************************
'  If first time through while loop
'     increment b1 by step value
'  else
'     increment b0 by step value
'  ********************************************
 if %1 = 1
  $b1 = r3c7 + $b1_step
  if $b1 < 0 then $b1 = 0.0001
 else
  $b0 = r2c7 + $b0_step
  if $b0 > 34 then $b0 = 34
 endif
 call step_function

'  ********************************************
'  If the goodness-of-fit statistic is less than
'     the goodness-of-fit from advance values
'     set minimum values from current values
'  else
'     decrement step size from advance value
'     and search in opposite direction
'     (b1 if first time through else b0)
'  ********************************************
 if $fit < r1c7
  r1c8 = $fit
  r2c8 = $b0
  r3c8 = $b1
 else
  if %1 = 1
   $b1 = r3c7 - $b1_step
   if $b1 < 0 then $b1 = 0.0001
  else
   $b0 = r2c7 - $b0_step
  if $b0 > 34 then $b0 = 34
```

```
  endif
  call step_function
'   ******************************************
'   If the goodness-of-fit statistic is less than
'      the goodness-of-fit from advance values
'      set minimum values equal to current values
'   Else
'      set minimum values equal to advance values
'   ******************************************
  if $fit < r1c7
   r1c8 = $fit
   r2c8 = $b0
   r3c8 = $b1
  else
   r1c8 = r1c7
   r2c8 = r2c7
   r3c8 = r3c7
  endif
 endif

'   ******************************************
'   If first time through while loop
'      set advance values equal to minimum values
'   ******************************************
 if %1 = 1
   r1c7 = r1c8
   r2c7 = r2c8
   r3c7 = r3c8
 endif
 %1 = %1 + 1
endwhile

'   ******************************************
'   Iteration complete
'   If minimum values less than base values
'      set advance values equal to step size plus minimum values
'      set current values equal to new advance values
'      calculate goodness-of-fit statistic for new advance values
'      set base values equal to minimum values
'      continue
'   Else
'      If advance values not equal to base values
'         set advance values equal to base values
'         continue
'      Else (minimum spanned)
'         halve step size
'         If step size greater than stopping condition
'            continue
'         Else
             display final values
             STOP project file
'   ******************************************
if r1c8 < r1c6
 r2c7 = r2c8 + r2c8 - r2c6
 r3c7 = r3c8 + r3c8 - r3c6
 $b0 = r2c7
' if $b0 > 34 then $b0 = 34
 $b1 = r3c7
```

```
 if $b1 < 0 then $b1 = 0.0001
 call step_function
 r1c7 = $fit
 r1c6 = r1c8
 r2c6 = r2c8
 r3c6 = r3c8
 %1 = 1
 jump iteration1
else
 if (r2c7 <> r2c6) or (r3c7 <> r3c6)
  r1c7 = r1c6
  r2c7 = r2c6
  r3c7 = r3c6
  %1 = 1
  jump iteration1
 else
   $b0_step = $b0_step/2
   $b1_step = $b1_step/2
   if (abs($b0_step) > 0.000001) or (abs($b1_step) > 0.00000001)
    %1=1
    jump iteration1
   endif
 endif
endif

%7 = r2c6
%8 = r3c6
%9 = r1c6

menu print 16 20 0 11 Final   B0 value :-
menu print 17 28 0 11 B1 value :-
menu print 18 33 0 11 Fit :-
menu print 16 41 12 11 %7
menu print 17 41 12 11 %8
menu print 18 41 12 11 %9
quiet on
repaint off
end
```

```
' ***********************************************************************
' ***********************************************************************
' ***********************************************************************
' Project file  - arsim.pf1
'
' This project file calculates the Bayes Factor for a set of failures
' as defined in the paper by Raftery [14]
'
' Input values
'   Column 1  -  failure number
'          2  -  time between failures
' Output values
'   Column 3  -  time since start of reporting period
'          4  -  Bayes Factor (calculated from previous failures)
' ***********************************************************************
'
quiet on
repaint off

' ********************************************
' Position cursor at Column 2
' determine number of rows
' set $total variable equal to number of failures
' ********************************************
@r1c2
goto lower-edge
$total = rows(r2:c2)
%1 = 1
 %1 = $total - 30

' ********************************************
' Initialise variables,
' determine
'   $s  -  the sum of times from start of period
'          to failure for all failures
'   $t  -  time since start of period
' ********************************************
while %1 < $total +1
 %2 = 2
 $s = 0
 $t = 0
 while %2 < %1 + 2
   $s = $s + (%1-%2+2) * r%2c2
   $t = $t + r%2c2
   %2 = %2 + 1
 endwhile

' ********************************************
' Initialise variables for integral function
' ********************************************
 $r = $s/$t
 $y = 0
 $h = 0.05
 $integral = 1
 $sum_integral = $integral
 @r1c3
 rc5 = $y
 rc6 = $integral
```

```
' ********************************************
' Calculate integral function using Simpson's Rule
'           (algorithm obtained from [29])
' step length set to 0.05 and
' stop criteria when stepwidth area is less than
' 0.0001 of total area
' ********************************************
 label start_iteration
   $y = $y + $h
   $integral = exp(-$r*$y)*power($y/(1-exp(-$y)),%1-1)
   $sum_integral = $sum_integral + $integral*$h*4/3
   cursor down
   rc5 = $y
   rc6 = $integral
   $y = $y + $h
   $integral = exp(-$r*$y)*power($y/(1-exp(-$y)),%1-1)
   $sum_integral = $sum_integral + $integral*$h*2/3
   cursor down
   rc5 = $y
   rc6 = $integral
   if $integral < $sum_integral/10000 then jump end_iteration
  jump start_iteration
  label end_iteration
  $y = $y + $h
  $integral = exp(-$r*$y)*power($y/(1-exp(-$y)),%1-1)
  $sum_integral = $sum_integral + $integral*$h*2/3
  cursor down
  rc5 = $y
  rc6 = $integral

' ********************************************
' Plot integral function
' ********************************************
 graphics generate ar black/white screen temp
 quiet on
' ********************************************
' Calculate Bayes Factor
'   from first failure to current failure
' ********************************************
 $bayes_factor = 0.6449 * (%1-1) / $sum_integral
 %1 = %1 + 1
 r%1c3 = $t
 r%1c4 = $bayes_factor
endwhile
end
```

REFERENCES

1. Stalhane, T. "Software reliability: A summary of the state of the art", University of Trondhiem Report no STF14 A88034, (October 1988).

2. Mellor, P. "Software reliability modelling: the state of the art", in Butterworth & Co. (Publishers) Ltd., 'Information and software technology' vol. 29, no. 2, pp 81-98, (1987).

3. Jelinski, Z and Moranda, P. B. "Software reliability research", in New York: Academic Press, Ed. Freiburger, W., 'Statistical computer performance evaluation' pp 465-484, (1972).

4. Littlewood, B. and Verrall, J. L. "A Bayesian reliability growth model for computer software", J. Royal Statistical Society vol. C-22, no. 3, pp 332-346, (1973).

5. Miyamoto, I. "Software reliability in online real time environment", in New York: IEEE, 'Proceedings of the 1975 international conference on reliable software' pp 194-203, (1975).

6. Lingren, B. W. "Statistical Theory", Third Edition, MacMillan 0-02-979420-x, pp 487 and 491, (1976)

7. Hooke, R. & Jeaves, T. A. " 'Direct search' solution of numerical and statistical problems" JACM, vol. 8, pp 212-229, (1961).

8. Brocklehurst, S., Chan, P. Y., Littlewood, B. and Snell, J. "Adaptive Software Reliability Modelling", in Elsevier Applied Science, Ed. Kitchenham, B. A. and Littlewood, B., 'Measurement for Software Control and Assurance', ISBN 1-85166-246-4, pp 263-277, (1989)

9. Scholz, F.-W. "Software reliability modelling and analysis", IEEE Transactions on Software Engineering, vol. se-12, no. 1, pp 25-31, (JAN. 1986).

10. Nagel, P. M. and Skrivan, J. A. "Software reliability : Repetitive run experimentation and modeling", NASA, Rep. CR 165836, (1982).

11. Nagel, P. M., Scholz, F.-W. and Skrivan, J. A. "Software reliability : Additional investigations into modeling with replicated experiments", NASA, Rep. CR 1272378, (1984).

12. Bittanti, S., Bolzern P. and Scattolini R. "Parameter Identification for Software Reliability Growth Models", Selected papers from the eighth IFAC/IFORS Symposium 'Identification and System Parameter Estimation', pp 1349-1353, (1988)

13. Yamada, S. , Ohtera, H. and Narihisa, H. "Software reliability growth models with testing-effort", IEEE Transactions on Reliability, vol. R-35, no.1, pp 19-23, (April 1986)

14. Raftery, A. E. "Analysis of a simple debugging model", Applied Statistics (Series C) Journal, vol. 37, no. 1, pp 12-22, (1988)

15. Jeffreys, H. "Theory of Probability", Oxford University Press, 3rd Edition, Appendix B, (1961)

16. Grady, B. G. and Caswell, D. L. "Software metrics: establishing a company-wide program", Prentice-Hall, ISBN 0-13-821844-7, (1987)

17. Shooman, M. L. "Yes, Software Reliability Can be Measured and Predicted", IEEE 'Proceedings of the 1987 Fall Joint Computer Conference - Exploring Technology: Today and Tomorrow', pp 121-122, (1987).

18. Kruger, G. A. "Validation and Further Application of Software Reliability Growth Models", Hewlett-Packard Journal, vol. 40, no. 2, pp 75-79, (April 1989)

19. Musa, J. D. and Ackerman, A. F. "Quantifying software validation: when to stop testing ?", IEEE Software, vol. 6, no. 3, pp 19-27, (May 1989)

20. DeMarco, T. "Controlling Software Projects", New York: Yourdon Press, ISBN 0-13-171711-1, (1982)

21. Kitchenham, B. A. and Walker, J. G. "A quantitative approach to monitoring software development", Software Engineering Journal, vol. 4, no. 1, pp 2-13, (Jan. 1989)

22. Chang, C. K., Fang, K. and Yang, J. C. "Applying software reliability models to decision support", AFIPS 'Conference Proceeding of the 1986 National Computer Conference', vol. 55, pp 229-236, (June 1986)

23. McCabe, T. J. "A Complexity Measure", IEEE Transactions on Software Engineering, pp 308-320, (Dec. 1976)

24. Jaynes, E. T. "Prior probabilities", IEEE Trans. Syst. Sci. Cybernet., SSC-4, pp 227-241, (1961)

25. Akman, V. E. and Raftery, A. E. "Asymptotic inference for a change-point Poisson process", Ann. Statist., Vol. 14, pp 1583-1590, (1986a)

26. Bartholomew, D. J. "Stochastic models for social processes", Wiley, 2nd Edition, (1973)

27. Graybeal, W. J. and Pooch, U. W. "Simulation: Principles and Methods", Winthrop Publishers, ISBN 0-87626-811-4, p 86, (1980)

28. Murdoch, J. and Barnes, J. A. "Statistical Tables for Science, Engineering, Management and Business Studies", Second Edition, MacMillan, ISBN 0-333-02584-9, p 17, (1978)

29. Dew, P. M. and James, K. R. "Introduction to Numerical Computation in Pascal", MacMillan, ISBN 0-333-32897-3, p 201, (1973)